UNIVERSITY OF OSLO
Department of Informatics

Consistency Checking UML Interactions and State Machines

Master thesis
60 credits

Bjørn Brændshøi
(bjornbra@ifi.uio.no)

01.11.2008

Abstract

As changes are made during a software development process, related artefacts and elements of the system specifications may quickly become inconsistent. Todays software projects often consists of a large number of artefacts and thus the job of keeping them consistent is very hard or even impossible to do manually by the developers.

The scope of the method presented in this thesis is to locate and display inconsistencies within Unified Modeling Language (UML) Interactions and State Machines, where the Interaction is the primary specification and the State Machine is the concrete, implementable specification.

We demonstrate a manual method for consistency checking of Interactions and State Machines that is implemented as a tool to assist the developer keeping the specifications consistent on-the-fly while modelling. The tool is integrated with the Eclipse platform and was empirically evaluated in a case study which results show that the tool helps the developer in keeping the specifications more consistent than the manual method, in less amount of time.

There is a significant win by this kind of work if the developer community regards UML as more valuable when they are assisted in keeping their specifications consistent, which is important in order to make good use of specifications within a development process. This work is usually done manually, if done at all.

Contents

List of Figures

List of Tables

If an elderly but distinguished scientist says that something is possible, he is almost certainly right; but if he says that it is impossible, he is very probably wrong.

Arthur C. Clarke

1 Acknowledgments

This thesis is submitted to the Department of Informatics at the University of Oslo as a part of my Master's degree. I would like to thank my supervisor, Øystein Haugen, for his valuable advice and feedback, and for continuously guiding me in the right direction. I would also like to thank the people at Sintef IKT and others who have given me feedback and hints on what direction to take.

— Oslo, 01.11.2008 - Bjørn Brændshøi (bjornbra@ifi.uio.no)

Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.

John Von Neumann

2 Introduction

This section introduces this master thesis and gives hints on what is to be expected, how the research has been carried out and what the goals were. The last part presents the structure of this document.

2.1 Motivation and background

Today, many projects, where large complex systems are being developed, struggle with keeping their specifications consistent. In practice, this is not an easy task due to the lack of CASE tools that assist in coping with this problem. The specifications usually need to be consistent both over time (during different development stages) in addition to during a single stage of development. Developers often tend to do these checks manually (if checking at all), resulting in lots of time spent on a job that possibly could be automated in addition to the fact that the manual method may not be equally correct as a tool-based checking routine.

In [11] they argue that one promising approach to reducing requirements errors is to apply formal methods during the requirements phase of software development. A formal requirements specification can reduce errors by reducing ambiguity and imprecision and by making some instances of inconsistency and incompleteness obvious. Formal analysis can detect many classes of errors in requirements specifications, some of them automatically. They also argue that tools that automatically perform checks like consistency checking can save reviewers considerable time and effort, liberating them to do more creative work.

2.2 Research goals

This thesis addresses the problems described in the previous section and describes a method for checking consistency of dynamic UML diagrams. This method is developed to adopt the various UML concepts that we encounter when doing such a consistency check. The work has resulted in an implemented version of this method which goal is to aid software developers in keeping their specifications consistent throughout the development process. This tool does the job that the developer normally would do, but faster and less error-prone than the manual checking. The tool is tested and evaluated in an experiment which gave valuable feedback on both the method and the tool.

The goal of this research is to achieve better understanding of the consistency checking domain while developing and implementing a method for doing this. In addition, the implementation of the method must be usable for the developer community in a way that i) the tool helps the developers keeping their specifications consistent in a faster and more correct way than the manual method, ii) the developers choose to actually try keeping their specifications consistent and iii) developers sees UML as more usable when their specifications can be kept consistent.

2.3 Research method

We introduce a method for checking UML interactions and state machines for consistency and compares this method used manually with the computerized usage which is possible after implementing the method in a tool. This comparison was done in an experiment which was a coalition of both direct and indirect observation of the method in question. The experiment opened up for direct imposed observation of potential users while using the method and tool which gave insight into the pragmatics of the tool. It was also an imposed indirect observation as the participants filled out a questionnaire at the end of the experiment which gave insight into the pragmatics of both the tool and the method.

We have adopted certain aspects of the evidence-based approach presented in [33]. Their approach for evidence-based software engineering - (EBSE) aims to improve decision making related to software development and maintenance by integrating current best evidence from research with practical experience and human values. We have tried to approach our work in a way that is similar to the five steps they propose in order to practice EBSE:

Defining an answerable question Is the consistency checking method implementable and does it improve the developers' effectiveness?

Finding the best evidence Using services that provides articles, such as the IEEE Xplore[1], ACM Digital Library[2] and the library of the institute[3].

Critically appraising the evidence Being critical to papers and articles, as the evidence related to software engineering is "fragmented and limited, not properly integrated and without agreed standards" compared to other domains, e.g., medicine.

Integrating the critical appraisal with SE expertise Listen to the industry and relate the work done to the needs of the developers.

Evaluation of the process Run an experiment within the context of the technology in question.

2.4 Thesis structure

This thesis is structured as follows. In section 3 we introduce the domain of this thesis where modelling, specifically the Unified Modeling Language (UML) and the concepts of consistency are main topics. This section is essential for further reading and understanding of this thesis. In section 4 presents the method used and evolved in this thesis together with discussions relating it to other similar work and methods. The section contains the main work done in this thesis and forms the foundation for the implementation of this method which is presented in section 6 along with the frameworks and environments in which it runs. The details of the consistency checker tool is presented in A and the tool was tested in an experiment which is presented in section 7 together with the results and discussions.

The thesis finish with some discussion regarding the method and tool develop in the contrast of other relating works and also some discussion of alternative approaches in section 8 and then the conclusions are made together with some thoughts regarding future work in section 9.

[1] http://ieeexplore.ieee.org/
[2] http://portal.acm.org/
[3] http://www.ub.uio.no/umn/inf/

The entire history of software engineering is that of the rise in levels of abstraction.

Grady Booch

3 Domain and definitions

This master thesis is written within the domain of modelled software development. The use of modelling languages when developing modern computer systems gives the developers the possibility of applying formal patterns and notation in an object oriented analysis and design (OOA/D) environment. Model Driven Development (MDD) has, during the last years of tool and technology development, evolved into a solution for developers giving them the ability to define a solution while creating artefacts that becomes part of the overall solution[1].

The models created when designing a system can be used for diagram creation to, e.g., communicate with different stakeholders and code generation while keeping the history of the project traceable. The models are usually graphically visualized to represent code syntax and domain concepts and structures in an intuitive manner, a picture is worth a thousand words. While code visualization is more concrete and directly implementable, domain visualization is more abstract depicting concepts at the business level within the enterprise. For a specific domain, one can create a Domain Specific Language (DSL) which is tailormade with artefacts and structures that are extracted from the domain of interest. This lets the developer express problems and solutions in a more clarifying manner than a general purpose language could. It also allows the developers to easier communicate with the business analysts to better pinpoint the specifications of the system artefacts with respect to the actual business in question.

I will use the general purpose modelling language Unified Modeling Language (UML) [41] in our work which is a part of the OMG Model Driven

[1] http://msdn.microsoft.com/en-us/library/aa964145.aspx

Architecture (MDA) [44]. UML is widely used by the industry and is regarded as a de facto industry standard. UML is an available technology from the Object Management Group (OMG). UML is currently in version 2.1.x which is the version that will be used throughout this master thesis.

Within the domain of model driven development there are several interesting aspects to which we could have devoted this master thesis. The one area of concern that is the subject of this thesis is consistency checking of specifications with some sort of analysis and handling of the inconsistencies found together with the development of a tool that realizes this method. This is the subject of this master thesis and will be discussed in-depth together with the presentation of the results from an experiment that was made using the implemented tool.

3.1 Systems, models and diagrams

The terms system, metamodel, model and diagram frequently used within the modelling domain. They are also used throughout this article and their use is defined here. I will mainly use the terms as they are used by OMG in the UML specification.

3.1.1 System

Kristen Nygaard[2] made a definition of a **system** [34] that we find convenient:

> A system is a part of the world that a person (or group of persons) chooses to regard as a whole consisting of components, each component characterized by properties that are selected as being relevant and by actions related to these properties and those of other components.

This definition depicts a system that suits a software engineer well although its definition is not solely for computer systems.

Within the domain of modelled software development the components, i.e., models and diagrams, will together make the whole. The properties of the components are different entities and artefacts in which each play a relevant part within the component.

3.1.2 Metamodel

The prefix *meta-* means *about* and is used as a prefix to a term that is an abstraction of another concept, e.g., metadata is data *about* data.

[2] http://en.wikipedia.org/wiki/Kristen_Nygaard

The metamodel defines a language for describing a domain of interest; it is the collection of concepts that are the vocabulary with which you are talking about this domain [3]. The metamodel is a description of the constructs and rules needed to build specific models within this domain of interest and is typically more compact than the model, i.e., "the model of the model".

3.1.3 Model

A model represents the entity being modelled in a way that all unnecessary details are left out; it simplifies the real world and focuses on different aspects of the entity. It is an abstraction of the real thing and an instance of the metamodel. An architect models a building where properties like interior colours and other details are left out, they are not needed for the model to represent the entity being modelled. On the other hand, details like landscape and window sizes are represented in the model because these attributes play an important role. A computer engineer models a computer system where properties like implementation details can be left out or abstracted at different levels. On the other hand, interaction between the system and its users might be an important view to include.

3.1.4 Diagram

A diagram graphically represents entities and their relationships. These entities can represent anything in the real world or more detailed, pure software concepts. It is common for an entity to be included in several different diagrams, where each represent a view of the entity being modelled [47]. The diagram is merely an instrument for the developer. The diagram elements are graphics showing representations of entities, their structure and behaviour.

3.2 The Unified Modeling Language (UML)

UML is a successor of an excess of other model notations and was introduced in 1997, having success within the modelling environments since. It originally focused on structural components of single applications, but have evolved to cover a wide range of content.

UML is a semi formal modelling language used by developers to specify, visualize and document models of software systems, including both their structure, design and behaviour [38]. As it is a language is has therefore both syntax and semantics. The abstract syntax of UML is specified in the UML metamodel [39] and the concrete syntax of UML is specified by thirteen diagram types in the Superstructure Specification [39]. The static semantics is given as a set of well-formedness rules expressed in OCL [43], while the

[3] Derived from www.metamodel.com

dynamic semantics is given in the Superstructure Specification for each element of the language using natural language, i.e., English. The diagrams defined by UML are divided into three categories: structure diagrams (static), behaviour diagrams (dynamic), and interaction diagrams (dynamic). This thesis will be working with interaction diagrams (sequence diagrams) and state machine diagrams which are part of the dynamic diagram types.

As computerized tools do not allow us to manipulate the semantics directly, what we work with on the paper or on the screen is a syntactic representation [24]. OMG has proposed UML 2.x with quite informal semantics. A complete formal semantics is needed to take full advance of the language in, e.g., automated tools. Several approaches has been made to define the formal semantics of UML interactions [52, 54, 35] and state machines [13, 25].

This thesis assumes a certain degree of prior knowledge about the different aspects of UML 2.1 core concepts. Documentation is available at, e.g., [39] and [47]. Although the UML specification specifies how the diagrams of UML look, there is no standard defined on how the metadata of these diagrams should be defined, so the visual representation of the diagrams may vary between different software tools. E.g., the UML specification defines the syntactic notation of a lifeline as:

> A Lifeline is shown using a symbol that consists of a rectangle forming its "head" followed by a vertical line (which may be dashed) that represents the lifetime of the participant. [..]

This gives the developers of graphical editors the freedom of visually showing a lifeline any way they want as long as the previous stated constraint is kept. Because of this, there might be problems when exchanging models with including diagrams between different software developing environments. To deal with this, developers can choose to use OMGs own diagram specification UML Diagram Interchange [40]. This graphical notation formalizes how to exchange documents compliant to the UML standard between software tools - how model elements are represented and visualized in diagrams. It extends the UML metamodel by a supplementary package for graph-oriented information while leaving the original metamodel intact. DI version 1.0 is compliant with the UML 2.x metamodel.

Another graphical notation that is available is developed by Eclipse, in their Graphical Modeling Framework (GMF) ([18]). This notation is quite similar, but still different, to DI and is used by the Eclipse UML2Tools ([16]). By having two big actors like OMG and Eclipse developing two different graphical notations could be somewhat unfortunate for the maturity UML modeling community as Eclipse has a big congregation while OMG has the responsibility for UML (and other standards). This discussion is not the goal of this thesis and will such be left as an area of problems for others to do research.

3.2.1 The meta-metamodel and metamodel in UML

The metamodel defines the abstract syntax of the language. It specifies model elements, e.g., class, attribute, operation and component. The UML metamodel is defined using a subset of UML notation and semantics. The scope of the UML metamodel is the UML specification while the scope of a model is the project using UML.

The metamodel is defined by a meta-metamodel which in reality is a metamodel specified using the Meta-Object Facility (MOF) [42]. MOF is a language specification, just as UML, for metamodelling and is used as a platform-independent metadata management foundation for OMGs Model Driven Architecture (MDA) [44]. By using MOF as the meta-metamodel it opens for easy model interchange between architecturally related languages, e.g, UML and Eclipse ECore [17]. The meta-metamodel is reflective, meaning that it is used to describe itself, resulting in no need for additional meta-layers.

OMG has defined UML using a metamodelling approach with a four-layer metamodel hierarchy. See an example in figure 3.1 on the following page.

3.2.2 Models and diagrams in UML

In UML, models and diagrams are loosely coupled, they are not formally defined by the UML specification but still perform two different roles of the UML language.

The diagrams are used to visually render the model elements. This means that a model could have any number of diagrams graphically representing [parts of] it.

According to the superstructure of UML [39] a model contain three major categories of elements: Classifiers (describes a set of objects), events (describes a set of possible occurrences) and behaviours (describes a set of possible executions). These elements are the subject of a model, not its content, thus a model does not *contain* objects, occurrences or executions.

In UML, a model is an instance of the metamodel, as seen in figure 3.1 on the next page. How models and diagrams are related and interact varies across the different modelling tools available. Some tie them very closely together while others have a distinct notion of both a model and a diagram. This is interesting because we need to know how changes to a diagram are reflected in the model and vice versa, and if elements in the diagram are constrained to exist in the model or not. Generally, it is advisable that the diagrams can only visualize artefacts that exist within the model. This problem is of interest to the subject of this thesis and will be discussed.

The work in this thesis is based on interactions, graphically shown in interaction diagrams, and state machines, graphically shown in state machine

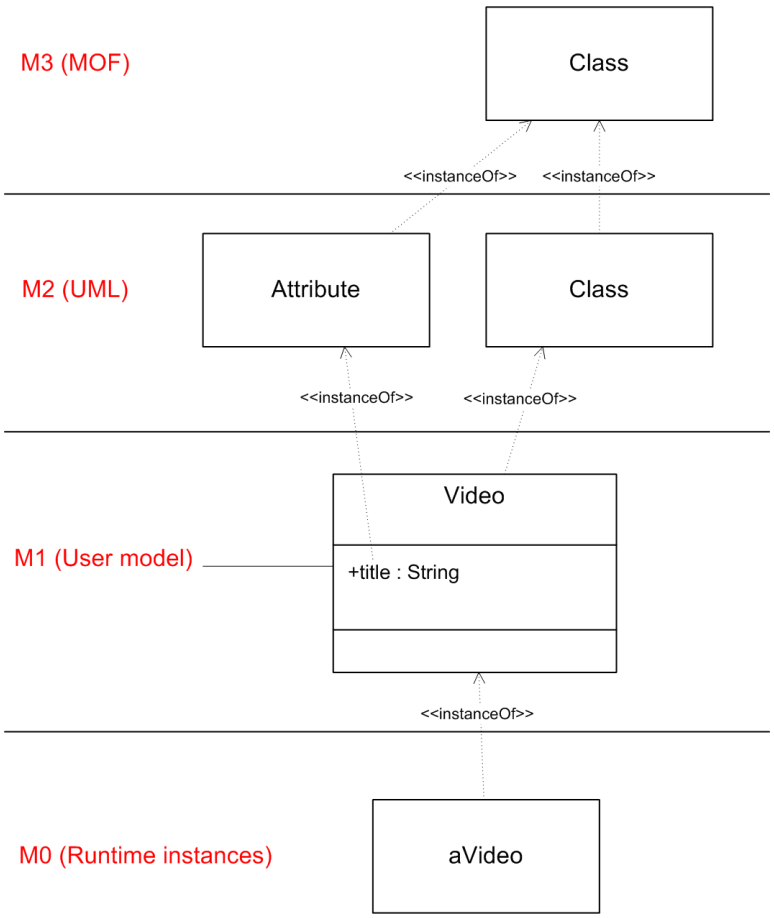

Figure 3.1: An example of the four-layered metamodel hierarchy (UML Infrastructure chapter 7.12).

diagrams.

3.2.3 Interactions

UML defines interactions as units of behaviour of an enclosing classifier. Interactions shows the flow of messages between participants of a system and can display the behaviour in several different types of diagrams but the differences between the types are not relevant to our discussion, so that we use the terms *sequence diagram* and *interaction* interchangeably.

An interaction consists of lifelines, messages and interaction fragments. The semantics of an interaction is given as a pair of sets of traces, i.e., positive (valid) and negative (invalid) traces. An interaction is often an incomplete specification due to the fact that it may have a set of incomplete traces (traces not described by the interaction) in addition to the positive and negative ones. We cannot know whether the incomplete traces are positive or negative. Negative traces can only arise from the use of either an **assert** or a **neg** combined fragment. Inconclusive traces are the ones that are neither positive or negative.

> **The semantics of an Interaction is given by a pair [P, N] where P is the set of positive traces and N is the set of negative traces.** $P \cup N$ **need not be the whole universe of traces.** [4]

A trace is a sequence of event occurrences. In this thesis, we are interested in looking at the traces of a given lifeline which is built up by interaction fragments and ordered by the implicit weak sequencing operator. Weak sequencing has the following properties:

1. The ordering of OccurrenceSpecifications within each of the operands are maintained in the result.

2. OccurrenceSpecifications on different lifelines from different operands may come in any order.

3. OccurrenceSpecifications on the same lifeline from different operands are ordered such that an OccurrenceSpecification of the first operand comes before that of the second operand.

The properties of weak sequencing often results in an interaction having multiple sets of traces and thus the task of checking a given lifeline with its corresponding state machine for consistency becomes more intricate.

An example of traces for an interaction with a negative set of traces showing the use of weak sequencing is shown in figure 3.2 on the following page.

[4] UML Superstructure 14.3.13

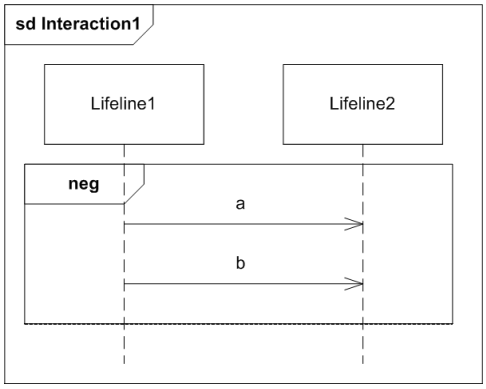

Figure 3.2: Interaction with negative traces

From this example, we get a set of negative traces (due to the neg combined fragment). When writing a trace, we write the sending of a message by writing its name proceeded by an exclamation mark, e.g., **!a**. The reception of a message is written by proceeding its name with a question mark, e.g., **?a**.

The following is the set of traces for Interaction1:

$$\text{Interaction1} = \{<!a,?a,!b,?b>,<!a,!b,?a,?b>\}$$

Weak sequencing results in the set of two traces of this interaction, i.e., both !a and !b may happen on Lifeline1 before ?a happens on Lifeline2.

As these message occurrences are all contained within a neg combined fragment, all traces are negative. The set of negative traces within an neg combined fragment operand is the union of the positive and negative traces. With the use of the semantic function $[\![\_]\!]$ [26] we show the semantics of Interaction1:

$$[\![Interaction1]\!] = \{(\emptyset, \{ <!a, ?a, !b, ?b >, <!a, !b, ?a, ?b > \})\}$$

Please note that the set of positive traces is the empty set $\emptyset$.

Challenges

As this technical report points out ([52]) there are several semantic paradigms that must be taken into account when dealing with concurrency within interactions. The UML standard is not even closely strict when it comes to this.

3.2.4 State Machines

The state machines we will be working with here are *behavioural state machines* that are used to express the behaviour of a system, as opposed to

protocol state machines, that are used to express the usage protocol of part of a system [41]. These state machines are finite state transitions based on the Harel statechart [12]. They are finite in the way that they have a finite number of states, transitions and actions.

Unlike an interaction, the state machine is a complete specification of negative and positive traces. The behaviour modelled in a state machine is accepted/positive behaviour for the system. All behaviour not modelled in the state machine is not accepted/negative behaviour. Thus, it describes explicitly positive behaviour and implicitly negative behaviour by saying that the behaviour modelled is accepted and all other behaviour is not. The state machine does not have the notion of inconclusive behaviour as the sequence diagram and is such a complete specification.

The building blocks of a state machine are vertices and transitions. The vertices are abstractions of a node in a state machine graph and are the source and targets of transitions [41]. Examples of a vertex are an initial state, state and pseudo state. The transitions are the directed relationships between the vertices, their edges.

A behavioural state machine consists of simple states, composite states and submachine states.

Simple states are states that the system enters while holding a certain invariant, but it can also model a dynamic condition where the state is a process of doing some task and the leaves when finished.

Composite states are states that has their own region(s) including sets of vertices and transitions. If it has more than one region it is called an *orthogonal state*.

Submachine state Semantically equivalent to the composite state, but specifies the insertion of the specification of a submachine state machine.

3.3 Approaching consistent specifications

As software is developed and modelling techniques are used, there can potentially be too many models and diagrams in a large project for the developers to have a clear overview at all times. Each model represents a view of the system in different stages of development or different time points within the same stage of development and they often have overlapping specifications.

It can be very important that the models and diagrams stay updated and in consistency with each other during the whole or parts of the development process, but we will discover that there are certain circumstances when inconsistency in the specification is wanted and the developers knowingly leave it that way.

The UML metamodel does not enforce semantic consistency; it is therefore up to the developers to ensure this whenever changes are made to the models, diagrams or requirements. A change such as one of these has to be reflected in other models and diagrams containing overlapping views of the system. As models and diagrams are used throughout the whole development process this is not something the developers will only encounter during the early stages.

It can be interesting whether this checking of consistency is possible to automate or not. E.g., if one has a model with an interaction and then a state machine that describes the behaviour of one of the lifeline in the interaction then maybe it is possible to have an algorithm for consistency checking of those specifications. If an inconsistency is found, then some form of analysis could be done to evaluate what actions could be taken and their consequences. We would probably also need to do some form of analysis on whether to take action towards the inconsistencies or not. This could be a process that runs in the background while the developer is modelling or be explicitly invoked by the user. It could be a stand-alone software or be integrated with existing modelling environment, such as Papyrus UML [46].

One first interesting problem to address is to decide where, if and how such consistency check is possible. There are similar work that has been done by others, as this problem is not new for the industry.

A second interesting area of problems is related to refinement of specifications. How does the consistency check notion relate to refinement and whether the two specifications considered to be consistent is a refinement of each other. This problem is discussed in section 4.7 on page 36.

3.4 Concepts of consistency

A problem with a modelling language like UML is the great possibility of creating conflicting specifications. As the UML offer multiple views of the same behaviour (e.g., behaviour specified in an interaction and state machine) it is easy to get in trouble with overlapping specifications when the same behaviour is modelled using multiple views and modelling notations. This can result in redundant (in the best case) and inconsistent (in the worst case) specifications [25]. The latter being the topic of this thesis.

According to [20], many classifications of inconsistency for the UML exists in literature today, i.e., several ways of defining inconsistencies in specifications made with UML. Today, few of these classifications are actually implemented in software development (CASE) tools.

These are the main sub-topics in this section:

1. Types of inconsistencies.

2. Consistency checking (or: Inconsistency locating)

3. Inconsistency analysis (if inconsistencies are found)

4. Inconsistency handling and its consequences (if inconsistencies are found and the user wants to take action).

These are all described in detail in the subsequent text.

There can be two different types of inconsistencies in a specification, horizontal inconsistency and vertical inconsistency.

3.4.1 Horizontal inconsistency

Horizontal inconsistency is inconsistent specifications at the same level of abstraction, i.e., at the same step in the software lifecycle. See figure 3.3 on the following page.

These inconsistencies are considered unwanted in most occurrences as the system being modelled tend to stay consistent in its behaviour during the same milestone in a software project. These types of inconsistencies can be more severe and thus more interesting to check for than the ones described in the next section.

3.4.2 Vertical inconsistency

Vertical inconsistency is inconsistent specifications on different levels of abstraction, i.e., in different stages of the development process. See figure 3.3 on the next page.

These inconsistencies are considered to be more usual than the horizontal ones, as going from one iteration/milestone in a software project to another often results in the system in development tend to evolve and thus the behaviour specified in earlier iterations do not necessarily have to be consistent with the current one. It is entirely up to the developer when doing the consistency checking if the model elements under test are representing behaviour from the same iteration or not. As a general rule of thumb, we suggest that checking for vertical inconsistency is less interesting for the developer than checking for horizontal inconsistency, but should not be ignored.

3.4.3 Consistency checking

The focus of this thesis is within the domain of the dynamic models in UML, interactions and state machines, that visually show behaviour of a system with their diagrams. Structural models like class diagrams can also be checked for consistency but this is not a part of the discussion in this thesis.

In the development process of a software system, typically some models and diagrams will change while others will not. Whenever the development process enters a new phase, new models are built which represent a system

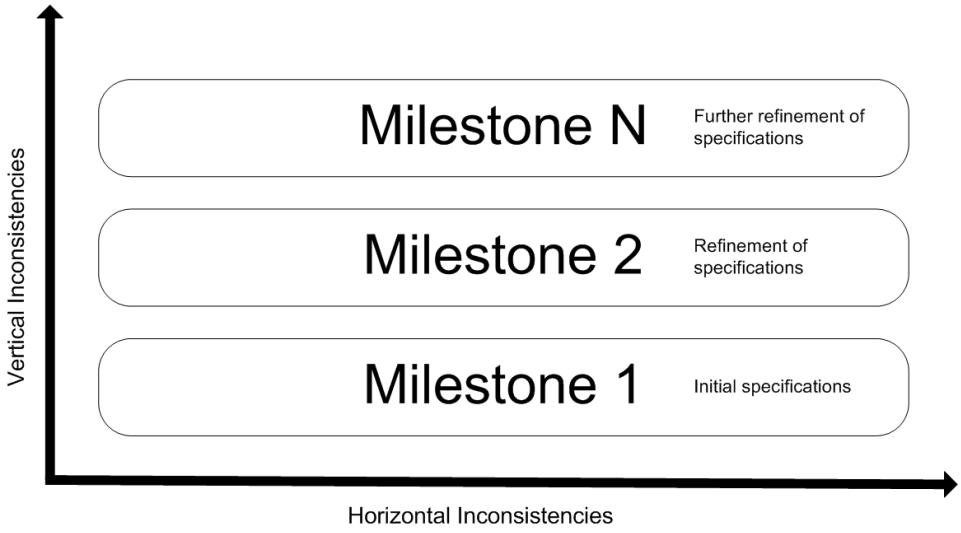

Figure 3.3: Horizontal and vertical inconsistencies

from a certain point of view and consists of different diagrams. It has to be determined if and how the different aspects represented in them fit together to express the behaviour of the objects involved. This concept is referred to as consistency. As UML defines a multitude of different model elements and diagrams the problem of consistency within those containing same elements of a model is important. As the models (and implicitly diagrams) change during a development process it gets harder to keep track of consistency. According to [28], the developers encounter two main problems of consistency during the process of model construction:

- Among different diagrams within the same model.
- Between the models.

The main focus of this thesis is not about diagrams as we deal with models when comparing for consistency checking. The models are considered to be the backbone of the diagrams. This discussion is continued in the next section.

This is how we define the consistency notion. Now we need a mechanism that checks whether a given consistency condition holds for a given model, this mechanism is introduced in section 4 on page 20.

Consistency checking - models vs. diagrams

Models and diagrams have their roles in the modelling process. Often, the developer will be modelling by using a diagram editor. This is easier than building a model manually using an model editor as it lets the developer

work with graphics that visually represents elements within the model, e.g., packages and state machines, rather than pure textual representations.

This thesis assumes that the *diagrams and models are tightly coupled*; all model elements in the diagrams must exists within the model, but not necessarily the other way around, i.e., not all model elements needs to be visualized in a diagram. This is important, because when we later will be talking about consistency, it does not really matter whether we think of the diagram or the model itself as the diagram elements represent the corresponding model elements.

This thesis is not about consistency checking graphical elements, but its about consistency checking model elements that can be represented by graphical elements.

It is important to note that this thesis is not about diagram comparison, the methods proposed are possible to execute without diagrams all together. The diagrams exist solely to ease the work of modelling for the developer by adding graphical data to the model elements for visual representation.

3.4.4 Inconsistency analysis

When the algorithm for consistency checking has found that a specification is not consistent, it will contain a number of inconsistencies. The inconsistencies are linked to a number of model elements, i.e., its scope, that are the source of the not corresponding behaviour. These model elements are expected to represent the same behaviour, but somehow does not.

The developer may choose to take action and need to know what consequences the different actions will have on those inconsistencies. This analysis will need to reason about these actions and predict the consequences and give the developer the information needed to choose whether to take action on the inconsistency or not.

In [5] they argue that to determine what action to take to deal with an inconsistency is determined by analysing the consequences that each alternative action has on the original specification. Based on this we can derive that the developers must get choices on how to deal with the inconsistencies and that a formal method alone cannot make such a decision without human interference [2].

Another form for inconsistency analysis is presented in [7] where they call it *impact analysis* which is an analysis done after the specification is verified as consistent by a consistency check. The method detects model changes and run an impact analysis to inform the developer of what impact changes has on the model. This is a variant of the inconsistency analysis presented above and is used to foresee the potential consequences of changes made to a model rather than looking to handle the already existing inconsistencies. It gives the developers the possibility for early decision making and change planning. This approach is a bit more complex and needs more algorithm

than the lazy variant described earlier.

3.4.5 Inconsistency handling

When modelling, you may run into inconsistencies in your models. These inconsistencies need to be handled, though not all inconsistencies are undesirable and not every undesirable inconsistency can be removed without creating new and possible more severe inconsistencies. The developer should be informed about such dependencies between inconsistencies as they are not independent events.

To handle an existing inconsistency the developer might choose to add or delete one or more pieces of information from the specification without necessarily removing the inconsistency all together. This may improve the overall situation and reduce the severeness of the inconsistency. In [5] they propose an approach to ensure that each action generates a new specification which, although may still leave the specification inconsistent, improves it in some way. This approach has the advantage of determining a course of actions based on the analysis of, and reasoning about, the consequences of possible alternative actions and runs tightly with the previous section where we presented the analysis of the specifications.

3.4.6 Living with inconsistencies

Although the inconsistencies are often unwanted, in software engineering there has long been a recognition that inconsistency is a fact of life [51]. Sometimes the consistency checking routine will report inconsistencies while the development is in a transient period of modelling where the results of the consistency check is not as important. Sometimes there might be certain circumstances where the developers are aware of the inconsistencies and they have some reason for not doing anything about them.

These inconsistencies must be tolerated and is equivalent to taking no action towards the inconsistencies.

3.4.7 Consequences

When taking actions against inconsistencies one will get consequences in form of a new specification. Whether this new specification is improved over the former or not can be derived from a defined set of desired consequences. The developer might find the new specification to have more inconsistencies than the former but the new inconsistencies might pose a less severe threat to the overall specification than the previous one(s).

The developer must evaluate the consequences of each action taken towards inconsistencies and rate them in a such way that whether the actions should be taken or not is easy to assess.

However far modern science and technics have fallen short of their inherent possibilities, they have taught mankind at least one lesson: Nothing is impossible.

Lewis Mumford

4 The consistency check routine

This section explain in detail how the consistency checking routine is developed, its prerequisites and outcomes. We will show examples on how this consistency checking routine can be applied, both manually and computerized.

There has been an incremental approach to the work done in this thesis and the software development. As the concept of consistency checking is a big task it is natural to begin with the simplest areas of problem and proceed with more complex situations later.

There is also a discussion on different possibilities on approaches and methods to develop such a consistency checking routine in section 8 on page 86.

The consistency check routine in this thesis looks for consistency in UML interactions (see section 3.2.3 on page 12) and state machines (see section 3.2.4 on page 13). These two kinds of specifications are highly dynamic and often model overlapping behaviour in a development project. The interaction typically includes one or more lifelines which represent some behaviour that is further, concretely specified in a state machine. There is a number of ways to check these two specifications for consistency, but also a number of limitations and prerequisites that has to be taken in account and dealt with. All which is the topic of this chapter.

4.1 Dynamic semantics comparison

The method that is applied in this thesis takes each interaction fragment on the lifeline, one by one, and looks for the corresponding externally observable

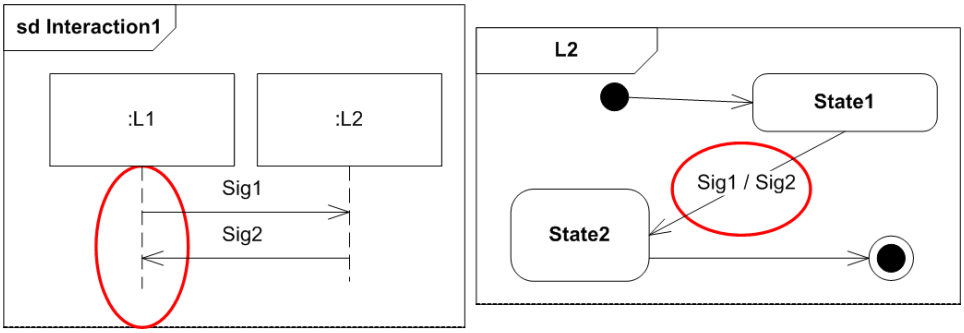

Figure 4.1: Semantic comparison of an interaction and state machine

behaviour in the state machine. This is a method that can easily be adopted to the white board community and "executed" manually. Technically, this method implicitly looks at traces but does not enforce the user to actually work with them in a manner that the method in section 8.1.1 on page 86 does, and it does not call for the transformation any of the models either.

By using this method, we define the lifeline as the primary specification and defines consistency as the need for finding the behaviour found at the lifeline within the behaviour specified in the state machine.

As an example, consider figure 4.1. Here we can do a semantic comparison by comparing lifeline L2 with the state machine. The state machine is supposed to model the behaviour found at the lifeline in a concrete manner and we check this by starting at the first interaction fragment, which is the reception of Sig1. The state machine will automatically enter the state State1 when executed and there it has an available outgoing transition which is triggered by the reception of the signal Sig1. So far, so good. Then we look at the next interaction fragment on the lifeline, which is the sending of Sig2. In the state machine, we are now situated at the transition which was triggered by Sig1 and this transition has an effect which is the sending of Sig2. Now, there are no more interaction fragments on the lifeline and we have proved the specifications to be consistent. If we did the same comparison with lifeline L1 we would immediately encounter problems.

The fact that we did not have any effects on the transition from the initial state, or that we did not end up in the final state is of no value for the dynamic semantics comparison. The behaviour modelled at the lifeline may only be partly the behaviour modelled within the state machine. This, together with several more prerequisites and rules are needed when doing this type of consistency checking. This is the subject of this thesis and the method will be throughly investigated and studied.

4.2 Prerequisites

There are certain prerequisites that need to be set and followed before a consistency check routine can run on a model. These are detailed below. First we give some definitions of terminology that is used extensively when talking about the consistency check routine:

Aligning lifeline and state machine Choose interaction + lifeline. Choose state machine + state. These positions represent the same point of execution. E.g., Interaction Example1 + lifeline a:A and State machine A + state Aidle.

Immediate Reachable Transition or state A transition or state that is reachable without having to go through transitions with triggers. E.g., going from one state to another via a junction pseudo state makes this latter state immediate reachable from the former[1].

Equal effect/trigger The effect or trigger must have an event with the same signal defined as in the interaction fragment in question and with the same receiving class if applicable.

Next vertex The vertex that is the target of the current transition.

4.2.1 Aligning the lifeline and state machine

When checking a given lifeline and a state machine for consistency, the lifeline and state machine must model overlapping behaviour. It is not possible to check for consistency in two independent specifications that do not include (parts of) the same behaviour and model elements. However, if they do, there is the possibility of the behaviour found within the state machine is an extension of the behaviour found in the lifeline specification (see figure 4.2 on the following page), as opposed to the two specifications modelling the exactly the same behaviour. Due to this, we need to align the lifeline with the state machine to tell the consistency check routine where the point of execution in the state machine is, that corresponds to the first event on the lifeline (see figure 4.3 on page 24). This is done by choosing what state in the state machine *aligns* with the first event of the lifeline.

This means that it is fully possible that the state machine models more behaviour than the lifeline, but not the other way around as the interaction is the main specification.

The alignment is done by choosing an interaction + lifeline and state machine + state.

The starting point for the consistency check routine is given by the alignment.

[1] Called "Compound transitions" in the UML Superstructure

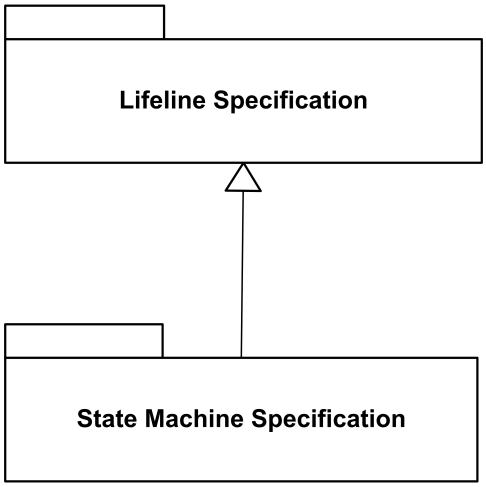

Figure 4.2: State machine extending the behaviour found at the lifeline

The alignment implicitly introduces the requirement that the lifeline and state machine must represent the same type as their property. E.g., lifeline **tomHanks:User** and state machine **User::StateMachine1** both represent the User property.

4.2.2 Transitions without triggers

The consistency check method does not allow a transition to be fired without a trigger. When the interaction fragment on a lifeline is the sending of a signal, the consistency check routine will look at transitions previously triggered (if in a state) for the corresponding effect. This means that if the transition is supposed to fire, the reception of the corresponding signal must occur on the lifeline prior to the sending. Consider figure 4.4 on the following page where the first event on the lifeline is the sending of Sig1. The transition from State1 to State2 does offer the corresponding effect, but it lacks a trigger. This means that the transition will never fire, and such the next event on the lifeline which is the reception of Sig2 that corresponds to the trigger on the transition from State2 to FinalState which will never be reached. If the alignment is done with L1 and State1, the consistency check routine must display an inconsistency that notifies that it cannot find an outgoing transition from State1 that has Sig2 as a trigger.

4.3 Limitations

The implementation of the dynamic semantic comparison method has some limitations which are presented in this section.

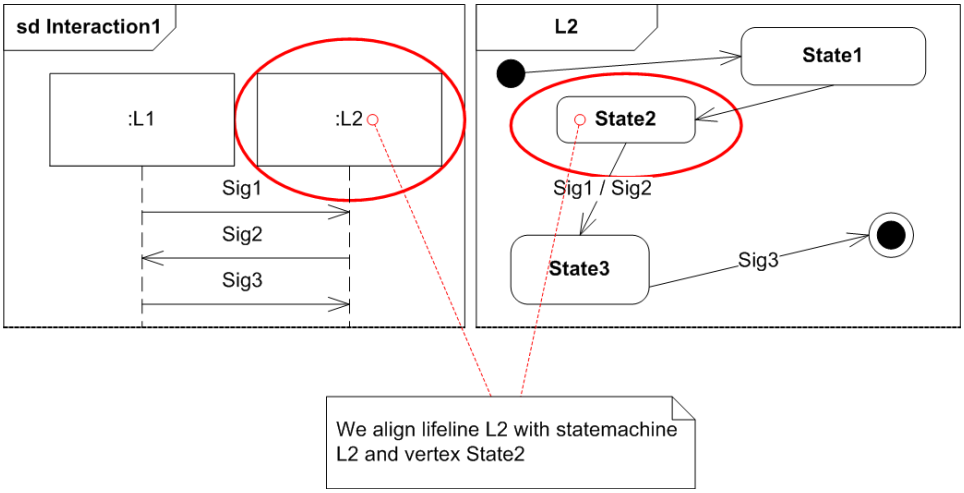

Figure 4.3: Alignment of lifeline and state machine

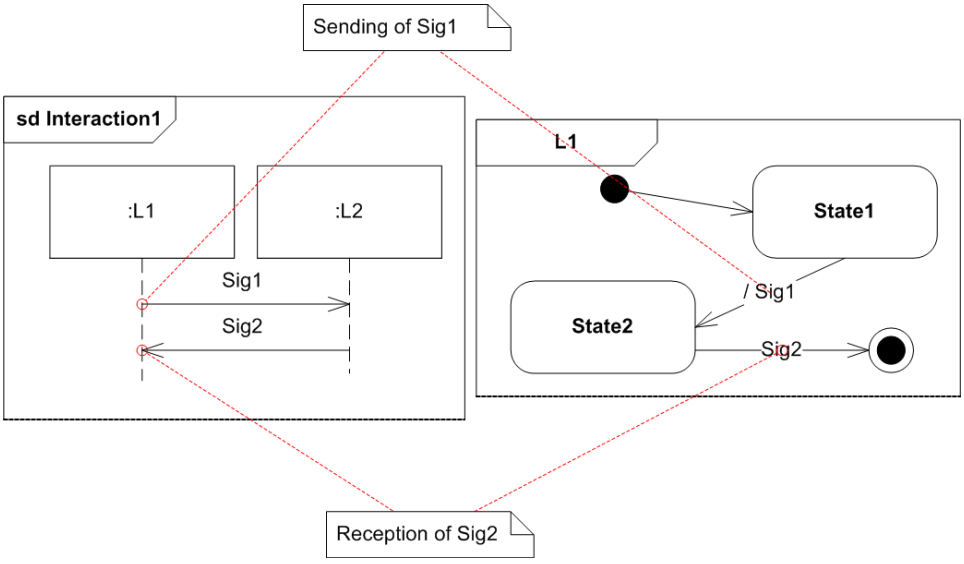

Figure 4.4: Transition without a trigger

4.3.1 Guards and constraints

The routine do not evaluate dynamic guards or constraints as these are runtime specific properties and are not possible to evaluate at the time of modelling [37]. This poses the restriction on the routine that we cannot, e.g., notify the user of guards that will always evaluate to false, neither can the routine choose transition paths based on the guards in junctions and choice points.

As an example, there could be transition paths that the consistency check routine sees as available but which are not at runtime due to guard(s) that evaluate to false. As figure 4.5 on the next page show, if the guard always evaluate to false when ran, the optional combined fragment will never run. This is not possible to know at modelling time.

This also poses the restriction that when there are several possible transition paths within the model but the guards will, at runtime, reduce this number and in such narrow down the possibilities. This is not possible to know prior to runtime and thus we are forced to create some deductive algorithms or consult the user to make a choice whenever more than one available transition paths is available.

One could implement some checking of the guards, like their textual equality and whether or not the variables defined (like *doOpt* in the figure) actually exists in the model. This is not doable without encountering several properties that makes the checking significantly more difficult. E.g., two guards may be equal in their results, even though they are textually different. Consider the two following guards, that are textually different but will always evaluate to the same result:

- `[doOpt == true]`
- `[doOpt != false]`

This example shows two guards that could be possibly be understood by a tool to evaluate to the same result, but it may not be that simple. Consider the two following guards that are not that simple to parse at modelling time:

- `[numIterations == MAX_ITERATIONS]`
- `[currentIteration == 5]`

For the sake of dynamic semantics comparison (see section 4.1 on page 20) we do not need to evaluate guards and constraints to be able to check the consistency notion of two specifications. The figure 4.5 on the next page shows an example of two specifications where guards are used. The consistency check routine does not need to evaluate the guards to determine whether the specifications are consistent or not. It does, however, notify the user of areas of problems, i.e., where there is lacking a guard with respect to the UML Superstructure [41].

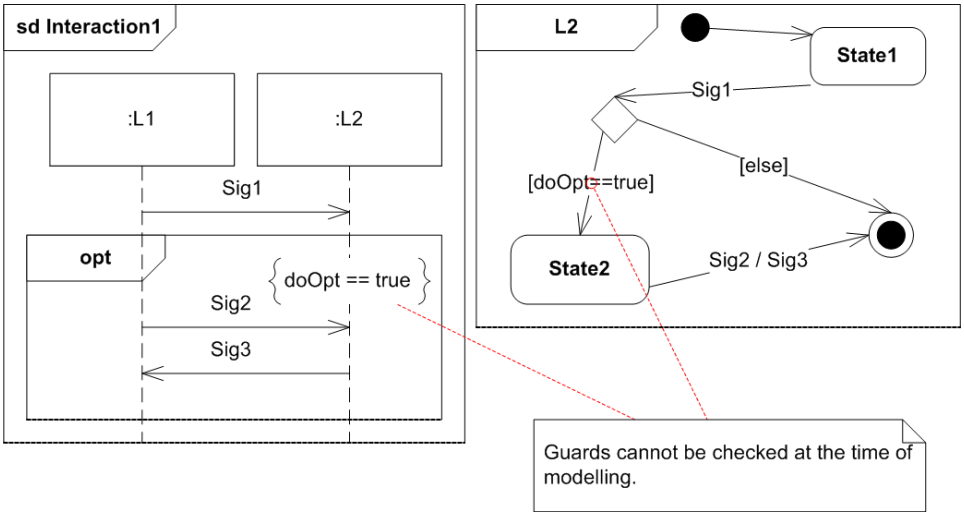

Figure 4.5: Guards within a combined fragment and a state machine

4.3.2 Arguments of signals

In interactions, signals sent have the option of having arguments, e.g.:

- **Sms("hotpos")**

These arguments can have variables within them that contain values that, e.g., affect guards on the lifeline. Even so, for the sake of the two specifications being consistent, the value of arguments on signal have been omitted for evaluation as they do not affect the result of the dynamic semantic comparison. In addition, the checking of arguments share much of the same problems that we encounter with guards and constraint as explained in section 4.3.1 on the preceding page.

4.3.3 Negative behaviour

One big challenge when working with behaviour of interactions and state machine is the fact that the state machine only models what it supposed to happen, i.e., positive behaviour, and does not model what is not supposed to happen, i.e., negative behaviour (see section 3.2.4 on page 13). Meanwhile, the interaction consists of both positive, negative and inconclusive behaviour (see section 3.2.3 on page 12). This poses a problem when the interaction contains operators that introduces negative traces, like **Neg**, **Refuse** or **Assert**. The use of negative traces is important, as the interactions are incomplete and are often underspecified. This means that there are often negative traces defined within an interaction.

When we are checking for consistency, we relate to negative traces by saying that the negative traces of the interaction defines behaviour that should not be included in the state machine.

The current consistency check routine do not deal with negative behaviour, thus this feature should be implemented in future versions, see section 9.3 on page 90.

4.3.4 Target of signals in transition effects

The sending of signals in state machines are done in effects on transitions. They can be defined using three different elements found in the UML metamodel. Two of these are slightly modified to be able to use in correlation with the JavaFrame framework (see section 5.1.5 on page 48).

The effects considered here are either an Activity or OpaqueBehavior. The activity is usually visualized in an activity diagram and supports the sending of signals by using the SendSignalAction element. This is the way UML suggest how signals can be sent using acitivies. As the implementation of the consistency check routine (see section 6.2 on page 55) is supposed to be implemented within a framework of tools for a specific course[2] we have implemented two other ways of sending signals. One is within the activity, by using a regular OpaqueAction and adding a body that corresponds to the Java code that sends signals which is used by JavaFrame:

```
output(sig, csm.to_icuproc, csm);
```

Where *sig* is the signal, *csm* is a pointer to the current state machine and *to_ icuproc* is a port in the composite structure that directs the signal to a given property (e.g., the state machine ICUProcess).

The same Java code can be added in an effect on a transition by creating an OpaqueBehavior, as previously mentioned, and adding a body in the same way as the OpaqueAction.

The restriction regarding these variations of sending a signal in a state machine is the target of these signals. The second parameter of the *output()* function is the name of a port within a composite structure that has an attached connector that leads to another port of another property. As the consistency check routine is restricted to the dynamic semantic comparison of interactions and state machines, we do not wish to extend the coupling with the JavaFrame framework further by also looking at this parameter which means working with composite structures as well.

Thus, we assume that the usage of the second and third parameter of the "output(..)" method is correct. Please note that when using the SendSignalAction, the receiver of the signal is defined in the activity and will thus be checked.

[2] INF5150 - held at the Department of Informatics, University of Oslo

4.3.5 Aligned state is a submachine

When the aligned state is a submachine, it imposes a challenge to locate the submachines' owning state machine; when the submachine terminates and the lifeline has more interaction fragments, the routine would have to know:

1. That the state machine terminating actually is a submachine.
2. What state machine to continue the consistency checking at, for the next interaction fragment.

This calls for the need of either limit the routine to only look at the top level state machine or tell the routine that "the state machine we start looking at is actually a submachine *within* ***this*** *state machine: [..]*".

In UML, a state machine can be nested by the submachine notion. They are stored stack-wise in order to know what the current state machine is and whether it has emerged as a submachine from some owning state machine. The routine may implement the same concept and by doing that knowing what state machine to continue looking into when facing this problem simply by popping the stack.

As an example, if the alignment is the following, as shown in figure 4.6 on the following page:

- Interaction: sd Interaction1
- Lifeline: Lifeline1
- State machine: **Submachine1**
- Vertex: Initial1

If the routine is not told that Submachine1 is a submachine (it could be named anything) and that its owner is StateMachine1, it would not know where to look for **Sig2**.

4.4 First version of the routine

When running the check routine we need a property that holds the current vertex (**CV**) of the state machine. As we run the lifeline sequentially, we do not need the same here.

The CV needs to be set before the routine will start if aligned on the initial state: If the first event on the lifeline is an incoming message, we set CV to the first state (the target of the outgoing transition from the initial state).

How to update CV when running the routine: When we reach an incoming message on the lifeline, set CV to the next vertex only if the next event

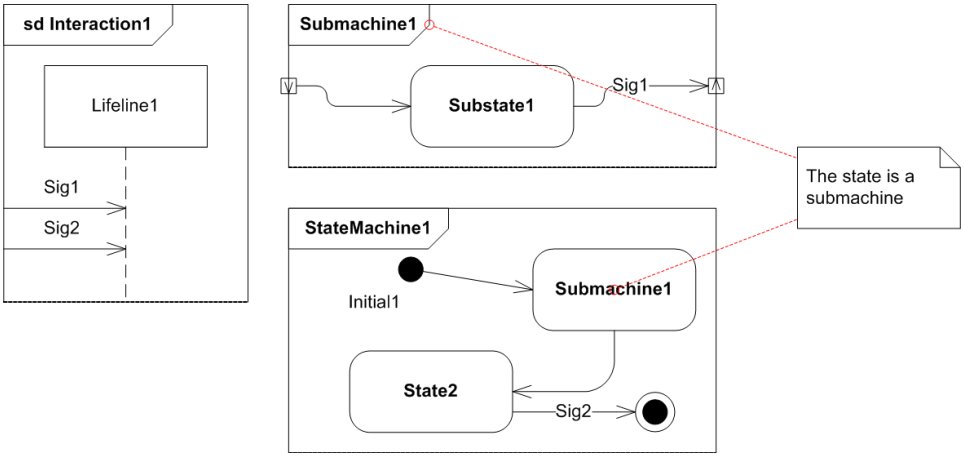

Figure 4.6: Example of submachine problem

on the lifeline is an incoming message. This is because when we have the trace <!s,?t> as found on lifeline b:B in Example1, this corresponds to one single transition having both a trigger ^s and an effect t, we must check for both before moving to the next vertex.

With this in mind, we define the second version of the consistency check routine:

Look at each lifeline in the interaction separately together with a state machine.

For each lifeline these rules must be applied for the chosen alignment to be consistent:

1. Align the lifeline L with the state machine (SM).
   (a) If aligned on initial state I:
      i. If first event on L is outgoing: CV = I
      ii. If first event on L is incoming: CV = I.transition.target
2. Look at each interaction fragment IF on the lifeline:
   (a) IF is incoming message: Look for an immediate reachable transition T with a corresponding trigger.
      i. If next interaction fragment is an incoming message: CV = T.target
   (b) IF is outgoing message: Look for an immediate reachable transition T with a corresponding effect.
      i. CV = T.transition.target
3. When the lifeline terminates: Look for an immediate reachable final state.

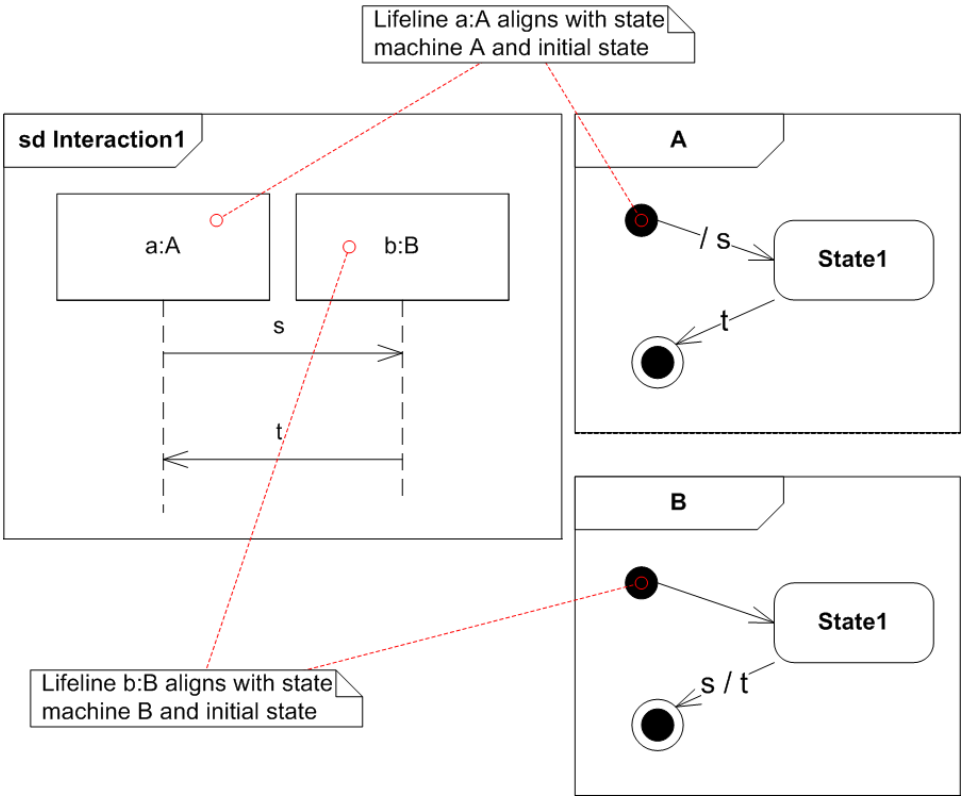

Figure 4.7: Example interaction and state machines

4.4.1 Example

Consider the example interaction and state machines shown in figure 4.7.

- First look at lifeline a vs. Statemachine A:

  1. First event on lifeline is outgoing: Initial state aligns with top of lifeline. CV = initial state. (1)
  2. Lifeline has outgoing message s: State machine has an immediate reachable transition with effect ^s. CV==initial state, therefore set CV = Aidle. (2b)
  3. Lifeline has incoming message t: State machine has an immediate reachable transition with trigger t. CV = final state. (2a)
  4. Lifeline terminates: State machine is in final state. (3)

Lifeline Example1-a (Figure 2) and state machine A (Figure 3) are consistent.

- Now look at lifeline b vs. Statemachine B:

1. First event on lifeline is incoming: Initial state aligns with top of lifeline. CV = Bidle. (1)
2. Lifeline has incoming message s: State machine has an immediate reachable transition with trigger s. Next interaction fragment is an outgoing message, stay in Bidle. CV = Bidle(2c)
3. Lifeline has outgoing message t: State machine has an immediate reachable transition with effect ^t. No change in current vertex. CV = Bidle.
4. Lifeline terminates: State machine has final state in immediate reach.

Lifeline Example1-b (Figure 2) and statemachine B (Figure 4) are consistent.

Conclusion: Interaction diagram Example1 is consistent with state machine A and B.

4.5 Second version of the routine

Until now, we dealt solely with messages along the lifeline. The next natural step would be to implement handling of combined fragments.

For this to be possible, we needed to add some functionality to the first version of the routine.

When we encounter a combined fragment (CF) on the lifeline, we only want to check the CF specific properties before we continue and let the routine look at the interaction fragments within the CF. Each CF has different properties that must be checked, these properties are defined in the UML superstructure [41]. The pointer to the current vertex might have to be updated according to what type of CF we encounter and how many operands there are. E.g. if we encounter an ALT combined fragment with two operands, then we should be able to find, e.g., a choice pseudo state with at least two outgoing transitions representing the two operands.

Second version of the consistency check routine:

For each lifeline these rules must be applied for the chosen alignment to be consistent:

1. Align the lifeline L with the state machine (SM).
   (a) If aligned on initial state I:
      i. If first event on L is outgoing: CV = I
      ii. If first event on L is incoming: CV = I.transition.target
2. Look at each interaction fragment IF on the lifeline:

(a) IF is incoming message: Look for an immediate reachable transition T with a corresponding trigger.

i. If next interaction fragment is an incoming message: CV = T.target

(b) IF is outgoing message: Look for an immediate reachable transition T with a corresponding effect.

i. CV = T.transition.target

(c) IF is combined fragment: Check combined fragment specific properties.

i. [CV = CV.transition.pseudoChoice]

3. When the lifeline terminates: Look for an immediate reachable final state.

4.6 Current version of the routine

The iterations leading up to the current version of the consistency check routine led to the following conclusions:

- We need to look at each interaction fragment, from the top to the bottom of the lifeline.
- We need to move "ahead" in the state machine whenever the reception of signals corresponds to a trigger on a transition.
- We need to be able to handle combined fragments that introduces several traces to the specification.
- We need to solve problems of non-deterministic behaviour whenever more than one possible transition path is available.

The focus of the behaviour in the interaction and state machine is on the overall behaviour. When there is an optional combined fragment on a lifeline, it does not matter how the actual state machine looks like as long as we can recognize the behaviour found on the lifeline both when including and excluding the behaviour within the combined fragment. In earlier iterations, the routine predicted too much about the internals of the state machine, e.g., when there is an OPT combined fragment there must be a choice pseudo state with two outgoing transitions - one describing the behaviour including the combined fragment and one describing the behaviour excluding it.

The current version of the routine takes into account that combined fragments introduces new traces and it starts a completely new consistency check instance for each trace. See figure 4.8 on the following page. This means that, e.g., whenever an OPT combined fragment occurs, we have one check

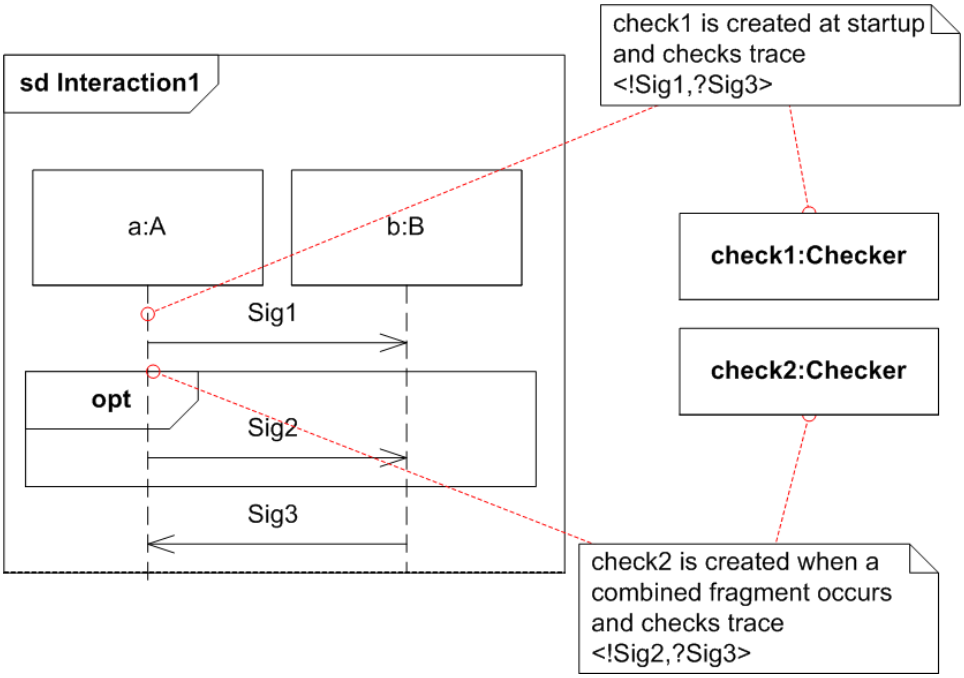

Figure 4.8: Traces of a specification with a combined fragment

instance that looks at the trace excluding the combined fragment (**check1** in figure) and one check instance that looks at the trace including the combined fragment operator (**check2** in figure). As all interaction fragments previous to the combined fragment has already been checked, we do not need to include those in the trace which **check2** is checking. Thus, when the optional combined fragment occur, the set of traces immediately splits and introduces the one including the combined fragment. The set of traces of the lifeline is, a:A = {<?sig1,?sig3>,<?sig1,!sig2,?sig3>}. The routine creates a new instance of the check routine for each operator in the combined fragment.

It also takes into account that the state machine could possibly be a subset of the behaviour modelled on the lifeline. This means that there is no need for either an initial pseudo state or final state.

We introduce some more concepts:

TriggeringTrans the current triggering transition within the state machine.

EffectTrans the current transition on which we are looking for an effect.

Current version of the consistency check routine:
For the chosen alignment to be consistent, these rules must apply:

- Align the lifeline L with the state machine SM.

- Current vertex CV = SM.first vertex

- Look at each interaction fragment IF on the lifeline:

- IF is **incoming message**:

  1. If TriggeringTrans is not NULL, CV = TriggeringTrans.target
  2. If EffectTrans is not NULL
     (a) CV = EffectTrans.target
  3. Look for an immediate reachable transition T within an available transition path from CV with a corresponding trigger.
  4. If no transition found: RETURN NOT OK
  5. If T has no effect OR next interaction fragment (skipping any cf's) is an incoming message: CV = T.target
  6. ELSE
     (a) CV = t.source
     (b) TriggeringTrans = T

- IF is **outgoing message**:

- If TriggeringTrans is not NULL

  1. If TriggeringTrans contains the effect RETURN OK
  2. ELSE start looking for effect at TriggeringTrans.target

- If EffectTrans is not NULL AND CV is a Pseudostate

  1. If EffectTrans contains the effect RETURN OK
  2. ELSE
     (a) If EffectTrans.target is a State
        i. CV = EffectTrans.target
        ii. RETURN NOT OK (we can not move further than this, as moving out of a state requires a trigger)
     (b) ELSE start looking for effect at EffectTrans.target

- Look for an immediate reachable transition T with a corresponding effect.

- If no transition found: RETURN NOT OK

- EffectTrans = T

- IF is **combined fragment**:

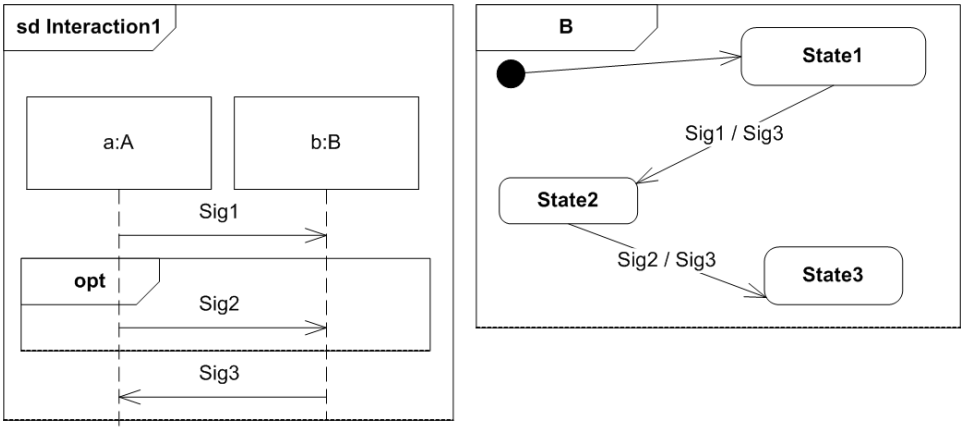

Figure 4.9: Example for current version of the routine

1. Save the CV in CV_rollback and TriggeringTrans is TS_rollback
2. For each operand:
   (a) Create new instance of the consistency checker on the trace including this operand AND the interaction fragments after the combined fragment.
   (b) CV = CV_rollback and TriggeringTrans = TS_rollback
3. Set the current interaction fragment on the lifeline to the one AFTER the combined fragment.

- IF is a **destruction event**: Look for an immediate reachable final state.
- IF is a **creation event**: Look for the creation of the represented property of the message target lifeline.

4.6.1 Example

Consider the example interaction and state machine shown in figure 4.9.

We align Interaction1 + lifeline b:B with state machine B + initial state.

The routine creates a consistency checker, **check1**, and executes it on this alignment.

- Current Vertex (CV) = initial state.
- **check1**: Interaction Fragment (IF) is incoming - Sig1:
  1. EffectTrans is null, CV is unchanged.
  2. State machine has an immediate reachable transition T from CV with Sig1 as trigger, transition State1->State2.

3. T has effect - CV = T.source (State1), TriggeringTrans = T.

- **check1**: IF is OPT combined fragment:

  1. TS_rollback = TriggeringTrans (State1->State2)
  2. CV_Rollback = CV (State1)
  3. Create new consistency checker, **check2**, that checks the following trace: <?Sig2,!Sig3>.
     - **check2**: IF is incoming - Sig2:
       (a) TriggeringTrans is not null, CV = TriggeringTrans.target (State2).
       (b) EffectTrans is null, CV is unchanged.
       (c) State machine has an immediate reachable transition T from CV with Sig2 as trigger, transition State2->State3.
       (d) T has no effect - CV = T.source (State2), TriggeringTrans = T.
     - **check2**: IF is outgoing - Sig3:
       (a) TriggeringTrans (State2->State3) contains the effect with Sig3. TriggeringTrans = null.
     - **check2**: Done.
     - CV = CV_rollback (State1).
     - TriggeringTrans = TS_rollback (State1->State2).
     - Set IF to the one after this combined fragment - Sig3.

- **check1**: IF is outgoing - Sig3:

  1. TriggeringTrans (State1->State2) contains the effect with Sig3. TriggeringTrans = null.

- **check1**: Done.

Conclusion: No inconsistencies found on this alignment.

4.7 Relating to refinement

Another aspect in addition to the consistency notion is whether the two aligned specifications are a refinement of each other. A specification is a refinement of another when it is transformed into a more concrete specification, reaching for the goal of being a completely deterministic implementable specification. In a typical software development process, specifications are refined either compositionally or stepwise by reducing their nondeterminism. As pointed out in section 3.4 on page 15, the consistency checking can be

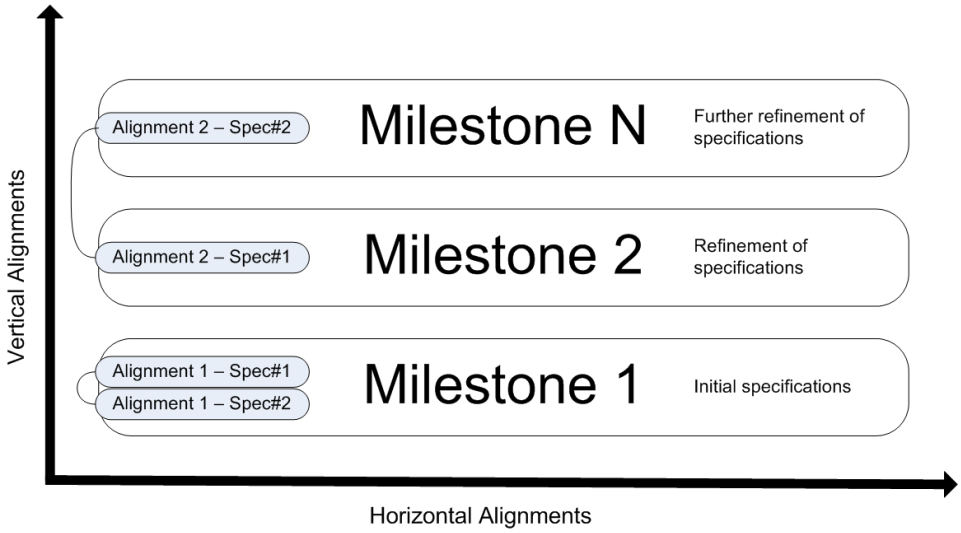

Figure 4.10: Possible alignments where *Spec#x* is either the aligned Lifeline or the aligned State Machine

done horizontally and vertically. This means that the two aligned specifications can be both in the same development iteration (horizontal) or in different iterations (vertical). This gives the possibility of checking for consistency given an alignment of a lifeline at an earlier iteration than the state machine, see figure 4.10. This opens up for a wide range of configurations for the refinement notion.

The prerequisite for the following discussion of refinement is that the specifications in question are consistent.

Intuitively, it could seem like the state machine is a refinement of the lifeline based on the fact that the lifeline is the primary specification for the dynamic semantics comparison method and the fact that the state machines may model more concrete behaviour than the interaction. It shows, however, that this is not true and it is highly dependent on a number of factors, e.g., the direction of the alignments' specifications. E.g., if the aligned lifeline and state machine are not of the same iteration in the development process, then the lifeline could possibly be a refinement of the state machine.

As the two aligned specifications' sets of traces may be a subset of the other, it is tempting to think of the notion of refinement as a bi-directional property, i.e., the behaviour modelled at the lifeline may be a refinement of the behaviour modelled in the state machine and vice versa. But, as the state machines is a complete specification which contains either positive or negative traces, the behaviour found at a lifeline can never be a refinement of the state machine as the set of traces for a lifeline will always include

inconclusive traces[3].

This leads to the conclusion that the only possible refinement given the two kinds of consistent specifications considered in the dynamic semantics comparison method is that of the state machine being a refinement of the lifeline. This is also probably the most convenient way of doing refinement in this setting as the developer is supposed to create state machines based on interactions in this setting. Also, state machines are often the source of model to text transformation, see section 5.1.4 on page 47, and thus it is the most complete specification.

In section 4.8 we use the STAIRS [26] method for refinement to see whether two consistent aligned specifications are a refinement of each other while keeping the consistency notion.

4.7.1 Refinement vs. consistency

Refinement and consistency checking of models are two concepts that may be important in order to facilitate the usability of UML for developers. UML is, unfortunately, often used only for its structural definition abilities, but the language offers much more.

By designing a software system using UML, one has the option of using several dynamic diagrams, in addition to the more "basic" structural ones, to specify both the behaviour of single objects and their interactions. If one makes use of this capability in coalition with the refinement notion while keeping the specifications consistent, there is a good chance that the whole development process sees new and improved aspects as opposed to a fractional, limited use of UML. E.g., there is a possibility of seeing increased productivity in a software project by using models throughout the process as it simplifies the process, letting non-technical stakeholders easier take part and enhancing the compatibility between systems. UML models can visualize program code and everyone knows that "*A picture says more than a thousand words*".

The concept of consistency can be seen as something that run in parallel with refinement. Specifications are refined to reduce the nondeterminism and enhance the concreteness. In addition, the developers should keep corresponding specifications consistent in order to maintain the quality of the design elements during the development process.

4.8 Relating to STAIRS

As interactions typically does not tell the complete story of a systems' behaviour, it can be a problem for developers seeking to create a complete

[3] This is not really the truth, by enclosing the whole interaction in an "assert" operator one eliminates all inconclusive traces, but this is a special case and is not considered further

specification of all possible behaviours of a system. UML is designed to allow underspecification as this is a feature for abstraction. One way of making a specification more complete is to do refinement. By refining a specification, one moves behaviour from the incomplete trace set to either the negative or positive set, where the goal is to reach a precise and detailed description applicable for formal handling. STAIRS [26] is an approach to refine interactions and defines three main methods for doing so:

1. Supplementing: Categorize inconclusive traces as either positive or negative.
2. Narrowing: Reduce the set of positive traces.
3. Detailing: Introducing a more detailed description without significantly altering the externally observable behaviour.

The STAIRS methodology define a number of requirements that it has been designed to fulfill:

- Allow both potential and mandatory behaviour.
- Allow both positive and negative behaviour.
- Capture the notion of refinement.
- Formalize the aspects of incremental development by supplementing, narrowing and detailing specifications.

Potential behaviour is expressed by using the ALT operator as defined in UML. Alternatives that are mandatory cannot be expressed by using the operators of UML. STAIRS has introduces an extension called XALT. This operator is used to express that there is not an option to exclude one of the alternatives. The use of XALT introduces traces residing in an interaction obligation (IO) for each alternative, and any correct implementation must support every IO.

Negative behaviour is expressed using the NEG operator of UML. This operator creates a negative trace and in combination with all traces leading up to it the negative fragment creates negative traces. Also, the subtraces following the negative traces are negative. The positive traces are the ones omitting the negative fragment(s).

Supplementing moves traces from being inconclusive to either positive or negative. Early specifications often lack completeness and may be refined during later stages of development. One may add functionality to a system by either add completely new scenarios, this moves inconclusive traces into the positive trace set, or one may add unwanted scenarios by moving inconclusive traces into the negative trace set. Supplementing does not alter the already positive and negative traces.

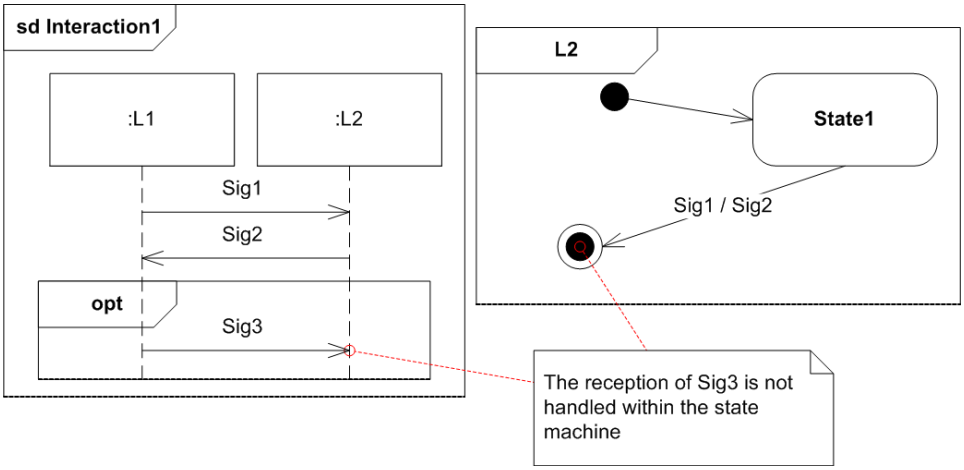

Figure 4.11: Possible supplementing of the State Machines' traces

Narrowing is done to produce a refined specification with less under-specification. This means to move traces from the positive trace set to the negative trace set. Narrowing does not alter the inconclusive or negative traces.

Detailing is done by using the decomposition mechanism of UML. One can detail an interaction by decomposing into more detailed views of, e.g., each lifeline. The behaviour of the specifications are the same, even though the detailed one contains much more detailed information. The sets of positive, negative and inconclusive traces are not altered by detailing.

To determine whether two specifications are a refinement of each other, we consider the mechanism proposed in STAIRS[26] to investigate the refinement notion of the two specifications aligned for dynamic semantics comparison.

- To ease the further reading, we hereby refer to the behaviour modelled at the lifeline for the *lifeline* and the behaviour modelled in the state machine as the *state machine*.

4.8.1 Supplementing

As supplementing involves making inconclusive traces either positive or negative while keeping the positive and negative traces intact. This would mean that either the lifeline or state machine must include behaviour that is inconclusive in its opposite specification and include positive/negative behaviour just as the source specification does. As a state machine is a complete specification and does not have any inconclusive traces, the only possible supplementing is that of the *State Machine supplements the Lifeline by including traces that is inconclusive in the set of traces for the lifeline.*

Consider the example in figure 4.11 on the preceding page. This example shows lifeline L2 with more behaviour than is modelled within the state machine. Does this mean that the lifeline is supplementing the state machine? The lifelines' set of traces are given as

$$[\![L2]\!] = \{(\{ <?Sig1, !Sig2 >, <?Sig1, !Sig2, ?Sig3 > \}, \emptyset)\}$$

The state machines' set of traces are:

$$\{<?\text{Sig1},!\text{Sig2}>\}$$

We see that the state machine is a subset of the lifeline:

- State Machine $\subset$ Lifeline

But this is not good enough to fulfill the refinement notion as the supplemented traces at the lifeline are not inconclusive traces within the state machine, i.e., the second trace of the Lifeline is not found in the set of inconclusive traces of the State Machine (which does not exist). As a consequence, the specifications are inconsistent due to the same factor.

If we consider the example in figure 4.12 on the next page, we see that the set of traces for the lifeline L2 is:

$$[\![L2]\!] = \{(\{ <?Sig1, !Sig2 > \}, <?Sig1, !Sig2, !Sig3 >)\}$$

The State Machines' set of traces are:

$$\{<!\text{Sig0},?\text{Sig1},!\text{Sig2}>\}$$

This is an example where the set of traces for the lifeline is a subset of the set of traces to the state machine, taken into account that Sig3 is within the negative trace of the state machine:

- Lifeline $\subset$ State Machine

This shows that the state machine supplements the lifeline as it include behaviour found in the inconclusive set of traces for the lifeline while the negative set of traces for the lifeline remains negative (i.e., not modelled) in the state machine. Further, as a consequence of this, the specifications are consistent as well.

We conclude that it is not possible for the lifeline to supplement a state machine and the specifications will be inconsistent as well, but there is the possibility of the state machine supplements the lifeline while the specifications are kept consistent.

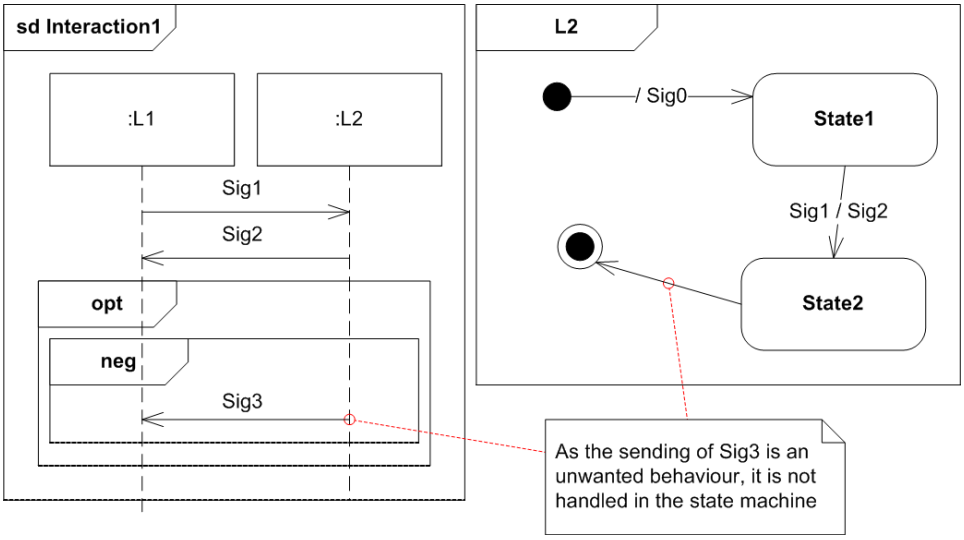

Figure 4.12: Possible supplementing of the Lifelines' traces

4.8.2 Narrowing

Narrowing means to move traces from the set of positive traces to the set of negative traces in order to reduce underspecification, while keeping all original negative traces intact. Relating this to the consistency notion of a lifeline and state machine would mean that either the lifeline is a narrowing of the state machine or vice versa. I.e. :

Lifeline is a narrowing of state machine Lifeline includes negative behaviour that is positive in the state machine.

State machine is a narrowing of lifeline State machine includes negative behaviour that is positive on the lifeline.

We will show whether this is possible or not.

Lifeline is a narrowing of state machine

Consider the example in figure 4.13 on the following page, we see that the set of traces for the lifeline L2 is:

$$[\![L2]\!] = \{(\emptyset, \{ <?Sig1, !Sig2> \})\}$$

The state machines' set of traces are:

$$\{<?\text{Sig1},!\text{Sig2}>\}$$

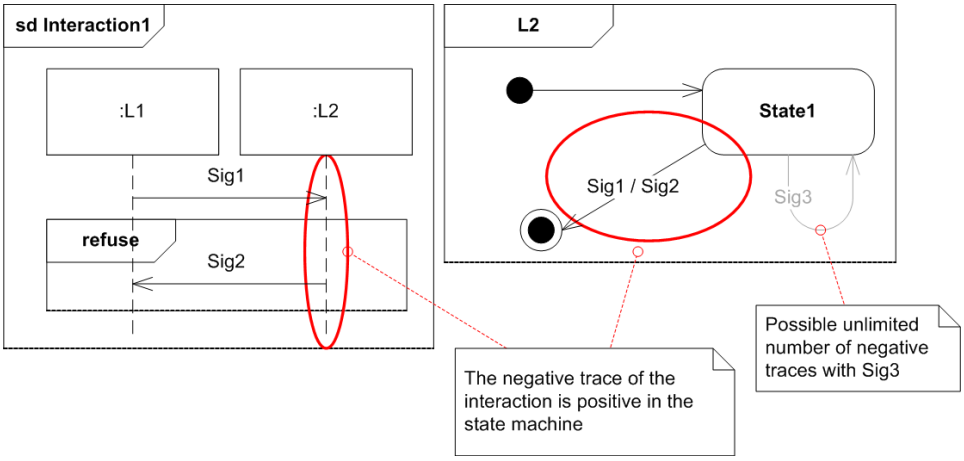

Figure 4.13: Lifeline is a narrowing of state machine

We see that the positive trace of the state machine is negative in the lifeline. The question now is whether the negative traces of the state machine are still negative. The negative behaviour of the state machine is everything that it does **not** do. The state machine is a *finite state machine* and has a finite scope, thus it has a finite number of signals which it can handle and it can not have an unlimited number of negative traces based on an unlimited number of signals. It could, however, have an unlimited number of negative traces based on a loop in the transition paths of the state machine based on one or more of the signals it handles, see the greyed-out transition which is triggered by Sig3.

For the refinement notion to hold, the lifeline would have to show all these traces as negative, which is fully possible as the number of negative traces for the state machine is finite.

Relating to the consistency notion, as the example shows, for the lifeline to be a narrowing of the state machine, the state machine has to act as the primary specifications and actually handle the traces which are negative in the lifeline. This is the opposite of our definition of consistency, and thus the specifications will be inconsistent.

Therefore, it is possible for the lifeline to refine the state machine by narrowing but the specifications will always remain inconsistent.

State machine is a narrowing of lifeline

Consider the example in figure 4.14 on the next page, we see that the set of traces for the lifeline L2 is:

$$[\![L2]\!] = \{(\{ <?Sig1, !Sig2>, <?Sig3>, <?Sig1, !Sig2, ?Sig3> \}, \emptyset)\}$$

The state machines' set of traces are:

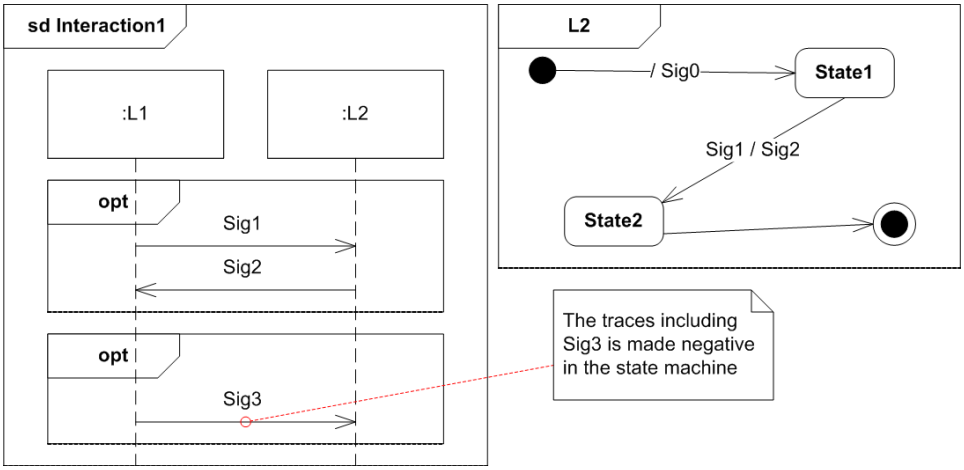

Figure 4.14: State machine is a narrowing of lifeline

$$\{<!Sig0,?Sig1,!Sig2>\}$$

We can see that the state machine has refined the lifeline by moving the two last traces of the lifeline into the negatives by not including that behaviour. This shows that it is indeed possible for the state machine to refine a lifeline by narrowing, but at the same time the consequences for the consistency notion is severe as the specifications are not consistent anymore.

4.8.3 Detailing

The refinement notion of detailing is about including more details without altering the positive and negative traces. This is done by refining an abstract specification into a more concrete one. Detailing is not trivial to apply when working with an interaction and a state machine.

When refining an interaction into another interaction by using detailing, one typically decompose a lifeline to reveal internal communications. In our case, one might say that the state machine can detail a lifeline by including more details regarding the same behaviour as found on the lifeline, we will look into this and see whether this statement holds . When checking for consistency, the lifeline is the primary specification and thus we do not allow the lifeline to include behaviour that do not exists within the state machine. Because of this, the lifeline can never be a detailing of the state machine.

Consider the example in figure 4.15 on the following page.

The set of traces for lifeline L2 is:

$$[\![L2]\!] = \{(\{ <?Sig1, !Sig2> \}, \emptyset)\}$$

The state machines' set of traces are:

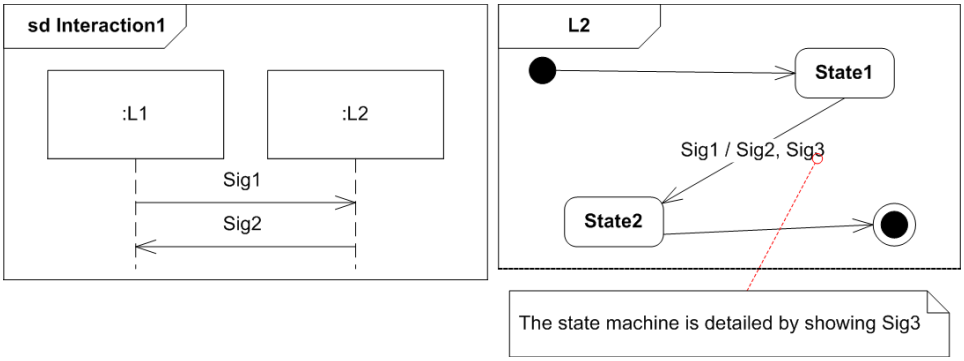

Figure 4.15: State machine is detailing a lifeline

{<?Sig1,!Sig3,!Sig2>}

In this example, the state machine sends Sig3 which is not modelled in the lifeline, which can be seen as a detailing of some internal communications that is not presented in the interaction. All positive and negative traces are intact as well and the specifications are consistent.

The man who removes a mountain begins by carrying away small stones.

William Faulkner

5 Related work

This section presents related work, such as the Model Driven Development approach in section 5.1 and other consistency related work in section 5.2 on page 49.

5.1 Model-driven Development

This section briefly explains the context of Model Driven Development - (MDD).

5.1.1 Why model-driven?

There are several reasons why many people are reluctant to integrate modelling in their system development process. Sadly, the "doodling phenomenon" still exists in people both at the developer side and also at the user side. The feeling of the models being "just paper" and the feeling of the job of developing models is just not worth it as the time spent could be used on writing code. They feel that "real" work has to be done with textual languages.

There is also a great deal of effort needed to be made in the area of models becoming out-of-date and inconsistenct with each other. In some cases, having an incorrect model is worse than having no model at all. The MDD approach attempts to solve these kinds of problems by expanding model creating past the design phase and integrating it into the implementation phase [1].

[1] Inspired by a presentation on MDD by Elizabeth Arrowsmith - CSE, UCSD 20/4/07

5.1.2 Model-Driven Architecture - MDA

MDA [44] is an MDD initiative created by OMG [2] since 2001. It defines a way of developing applications and writing specifications based on a platform-independent model (PIM) of the application of specification's business functionality and behaviour. Then the PIM model is automatically translated into platform-specific model(s) (PSM) for the desired target platform(s) with a tool.

UML is usually though of as the basis for MDA but a model needs to be MOF compliant to be labeled "MDA Compliant". Since UML is based on MOF, modelling in UML makes the models available for use in a MDA project, but any other domain-specific language (DSL) based on MOF will work. The reason for this is that the metadata must be understood in a standard manner which is a precondition for any ability to perform automated transformations between a PIM and PSM(s).

5.1.3 Model transformation

Today, with more and more abstraction of business processes and implementation details, we make models on several levels, see figure 5.1 on the next page. MDA defines three levels - ranging from the computational-independent model (CIM) through PIMs down to the PSMs. They each concerns a different audience, CIMs are aimed at business analysts working with business functionality. PIMs are aimed at system analysts for describing system logic for the CIMs. PSMs are aimed at system designers and provides additional elements specific to given deployment platforms.

There is a need to transform between models on different levels and also between models on the same level. This is possible by using the different tools for model transformation that exists, like ATL[3] (model to model) and MOFScript[4] (model to text).

5.1.4 Code generation

One of the major goals of model-driven development (MDD) is to be able to transform the models into executable code. By using tools for model to model transformation on CIM-PIM-PSM level and then tools for model to text transformation from the PSM models one can achieve this goal. How much code that is generated from the models vary from simple code skeletons to complete application code. There is trade offs for each level of code completion one strives for, but as a general rule of thumb one can be sure of the more code one wants to automatically generate, the more complicate the models and tools needs to be. As a positive effect, one gets the problem

[2] www.omg.org
[3] www.eclipse.org/m2m/atl
[4] www.eclipse.org/gmt/mofscript

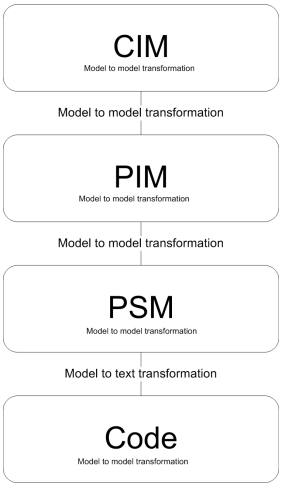

Figure 5.1: From computational-independent to platform-specific models

of models being out of date and inconsistent solved more or less for free as the models and code are tightly coupled.

5.1.5 JavaFrame

JavaFrame [27] is a framework for Java enabled modelling. It is a tool for creating Java code from state machines modelled in UML. JavaFrame is used in conjunction with the tool developed for consistency checking, and is the reason why there are some JavaFrame-specific functionality implemented. This is explicitly notified whenever it applies in this thesis.

The JavaFrame framework was developed as an answer to the fact that there was a need for producing effective Java code from UML. The goal of JavaFrame is "to improve the way the developers work by letting their programs become rapidly made according to specification, have high quality, be efficient, maintainable by competent people and adaptive". It works by letting the developer use UML to define composites with properties of state machines which communicate through ports and mediators. This lets the developer create a almost ready to run program by UML modelling, some minor Java coding is still needed to define the structural behaviour of the program, which is defined in the coding rules of JavaFrame.

In the setting of this thesis, the state machines used by JavaFrame should be specified by interactions and checked for consistency in order to maintain a correct and proper system specification.

5.2 Other consistency checking frameworks

In this section we discuss other frameworks that does a analysis, validation and verification of software systems.

5.2.1 SPIN (and PROMELA)

SPIN [22, 23] is a verification system that checks for logical consistency of process interactions, abstracting as much as possible from internal sequential computations developed at Bell Labs. The specifications which SPIN attempts to verify are written in a high level specification language called PROMELA (a PROcess MEta LAnguage).

PROMELA is a verification modelling language that provides ways of making abstractions of distributed systems. The language has the ability to define processes dynamically which communicates via message channels either asynchronously or synchronously and its intention is to make it easy to find good abstractions of systems design. It is a system description language, rather than an implementation language. The language is designed to target *software* systems rather than *hardware* systems, i.e., there are no notion of time or floating points. PROMELA offers the following language constructs:

- The creation and execution of processes (statically and dynamically) which are global objects.
- A set of data types like bit (1bit), bool (1bit), byte (8bit) and int (32bit) which are either global or local to a process.
- The creation and usage of message channels which passes messages in FIFO (First In First Out) order, which are either global or local to a process.
- Control flow structures, like case selection (if..[else]), repetition (loop) and unconditional jumps (goto).
- A construct for defining a sequence of code atomically.
- An assertion construct that produces an error when used by SPIN for verification.
- Complex data structures.

SPIN works in two basic ways, one is to use the tool to construct verification models that can be shown to have all required system properties. When the design has been proven sound, it can be implemented with confidence. Secondly, one can start from an implementation and convert critical parts of the system into verification models. There are tools for converting

C and Java programs into SPIN models. SPIN reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes and works on-the-fly.

SPIN has two different modes to operate in, simulation and verification. Simulation gives the developers some valuable insight in the model being built and is useful for debugging, until confidence is gained that the design behaves as intended. Verification is then what is applied to prove a specifications correctness. For SPIN to be able to verify a system specified in PROMELA, special language elements are used to make meta statements about the semantics of the model. These elements are assertion statements, special labels, never claims and trace assertions, e.g., safety properties can be checked by finding where an assert statement can fail. These meta statements are interpreted by the verifier. SPIN use the simulator to display the error trace whenever a counterexample to a correctness claim is encountered. When this happens, the verifier will then write the execution sequence to a trail file which the simulator can read to recreate the execution path. SPIN also has a graphical user interface called XSpin.

5.2.2 Telelogic TAU/Architect

Telelogic TAU[5] is a UML 2.1-based model driven development environment and the TAU/Architect package offers model validation and verification. The model verifier compiles and executes UML models created and provides and executable trace of the system. In addition it offer a simulation functionality to execute the system at an early stage in order to correct design errors as early as possible.

In contrast to the open source framework SPIN, discussed in section 5.2.1 on the preceding page, Telelogic products are commercial and thus finding detailed information on how the verifier and simulator works is not trivial.

5.2.3 Others

Alexander Eyged has done multiple publications on the subject and also works on an automated UML analyzer tool [3] amongst other model tools. His work may be interesting because of his methodology for instant consistency checking. In [2] he argues that a tool cannot automatically resolve inconsistencies in UML models because the tool cannot know whether an inconsistency is tolerable or not and why it was caused. Humans are needed to make decisions based on, e.g., gut-feeling, tradition, restrictions on the domain; aspects which is not possible for a tool to know about. The tool can therefore be used for presenting the inconsistencies to the user and predict what side effects repair actions on the inconsistencies will have.

[5] http://www.telelogic.com/

A. Eyged defines consistency rules that must be satisfied for the models to be consistent (valid). A consistency rule say something about the model, like that a message name in a interaction diagram and a state transition in a statechart diagram must have a corresponding method in the class diagram. It can be thought of as a condition that evaluates a portion of a model to a truth table [1]. Also, the time used to check for consistency is affected by the number of rules as it needs to apply every rule on every consistency statement. Therefore, the more rules needed to validate the more memory usage the procedure will need. Different projects may have different consistency rules, depending on the surroundings, constraints and environment for the software.

Consistency rules are also defined in [7] as what inconsistencies can be automatically modelled and detected by. They argue that each rule corresponds to one type of inconsistency and can be described informally or formally in any declarative language like OCL [43]. They have identified 120 consistency rules, which may increase or decrease as one gain experience while using the method presented. This may be a simple and straight forward method for defining a set of rules for declaring inconsistencies.

A. Eyged and others [7] makes use of a term that introduces an important factor about consistency checking; impact analysis. This is the notion used for the concept that whenever an inconsistency is found we generally need to know what happens if we correct it - its side effects. Inconsistencies are not independent events - fixing one might cause one or several new ones trigging reanalyzing or retesting. This is certainly not trivial because we often have many model elements that depend on each other and making change in one result in an unwanted ripple effect. The developer should always know about the resulting side effects prior to applying repair actions as it would have been a great help in planning ahead.

The approach introduced by A. Eyged [2] lets the developer see through all of the inconsistencies and choose whether to take action or not. Sometimes it is desirable to have inconsistencies (though mostly temporarily) the tool should therefore not enforce repair actions. This is the notion of tolerable and intolerable inconsistencies mentioned further up.

The instant consistency checking methodology as A. Egyed (and others) has developed [2, 1] has the advantage of being instant; meaning that it uses only a few milliseconds to validate each change made to a model - giving the modeller design feedback instantly while she is modelling. Other methods like the more general xLinkIt [10] which does consistency check on all kinds of documents and ArgoUML [48] does not have this instant factor on their validating. The xLinkIt is somewhat related to the method they have developed in [55] where they use the OMG XMI [45] to transform the UML model into XMI which is an XML document and then applies their own technique to add semantics to XML documents by attaching semantic information to the XML tags. The XMI document is then checked for consistency. Another

project that strives for consistency within XML documents can be seen at [4] in which they do research to support consistency of distributed documents on the Internet. They define relationships that are required to hold among the documents which are expressed through consistency rules and the elements related by those rules are associated through hyperlinks, named consistency links. The consistency rules are expressed used consistency syntax, which also is based on XML, and can be created using an editor.

In [32] they use description logic (DL)[6] in order to check the uml models for consistency. In order to ensure usability, their inconsistency detection and solutions is an user activated process, so the user is always in control and the implementation of the tool is simpler. They argue that automatic inconsistency detection would imply developing predicate application strategies, and that the development of this type of heuristics is still an open problem. Their tool, MCC (Model Consistency Checker) is a tool that provides UML 2.0 model checking reasoning using a DL implementation and it is implemented as a plugin for the Poseidon for UML[7] tool. The tool has a set of predefined inconsistency types that it looks for within the given models.

In [30] they introduces horizontal consistency as intra-model consistency and vertical consistency as inter-model consistency. This is also seen in [37] where they present different important issues when working with consistency:

- Definition of consistency
- Relationships between consistency and development process
- Approaches to check consistency
- Checking tools (where the two main approaches are checking directly the UML model and translation of the UML model into a formal language and use tool for performing checks on the target language)

Definition of consistency can be done by adding constraint or some form of well-formedness rules and the definition of inconsistency is whenever the model violates the added rules. The refinement notion can be applied to enforce the vertical consistency by applying constraints. For the models to be consistent, there must exist a specification in the set with respect to a refinement relationship. For the translational definition, the models is translated into a target language and then needs to satisfy some good properties to be consistent. This approach only applies to class diagrams, object diagrams and state machines. They argue that the model consistency should be preserved through refinement during a development process and that the models should be consistent with the development methodology.

In [9] they argue the need for objective means for establishing the quality of UML models as models generally contain a large amount of defects. They

[6] http://dl.kr.org/
[7] http://www.gentleware.com/

have developed a quality model for UML which aims to enable identifying the need for actions for quality improvement in the early stages of the system development. Early actions are less resource intensive and less cost intensive than later actions.

The work done in [50] shows validation of specifications, made by UML interactions and state machine, on the class level. This is different from our approach as we are looking at the dynamic semantics at the instance level where the participants of the interaction operate as a specific instance of an object. At class level, the participants of the interactions operate as generic instances of classes. They translate both the state machines and interactions into Petri Nets[8] which are then used in conjunction with each other to create a complete model of the system behaviour. On this complete model, they now compute certain logical properties to validate, e.g., "Compare the actions that are included in the shortest path of the (full model net) **LS** from the initial marking $\mathbf{M}_0$ to $\mathbf{M}_{end}$ with the actions that are modeled in the (interaction net) $\mathbf{L}_{interaction}$".

[8] http://www.informatik.uni-hamburg.de/TGI/PetriNets/

In the end we will listen to the voice of the machines. We will have to. There is no choice. We will not go back to tallow dips while the great shining wheels are there to bring us light.

Mary Heaton Vorse

6 The Consistency Checker tool

This chapter will explain briefly how the tool implementing the dynamic consistency checking method was developed and what software packages it relies upon. See appendix A on page 98 for details on the structure of the tool.

The consistency check routine in this master thesis was implemented within the Eclipse[1] development framework as a plugin.

A plugin is a complete stand-alone package that lets developers add, remove or change functionality in the Eclipse workbench (cf. [31]).

The consistency checking routine was implemented as a plugin that adds dedicated functionality to the Eclipse workbench for the consistency checking of interactions and state machines. The functionality lets the user align interactions with a state machines and it will then run a consistency check routine on the alignments whenever changes are made to its aligned elements. The plugin is both action- and view based, as it creates a new view with its own actions (e.g., *About* or *Help*).

This plugin is made available through an Eclipse update site at the following location:

http://www.bjornbra.no/umlconsistency/update

Follow these steps to install the plugin:

1. Select **Help** > **Software Updates** > **Find and Install**. Select **Search for new features to install**. Click **Next**.

[1] http://www.eclipse.org

2. Press **New Remote Site**. Choose an appropriate name, e.g., UML Consistency Checker. Type in the above URL. Click **Ok**.

3. Check the newly added update site in the list. Click **Finish**.

4. Check the **UML Consistency Checker Feature x.x.x**. Click **Finish**.

5. You will need to verify the feature installed. Click **Install All**.

6. You will need to restart the workbench.

The tool is now available in the workbench and can be activated by selecting **Window** > **Show View** > **Other** > **UML Consistency** > **UML Consistency Checker**.

6.1 The Eclipse platform

Eclipse [15] is an open-source platform for building integrated development environments (IDEs). This means that the developer can design her very own IDE by using the Eclipse backbone platform and any number of plugins that offers the functionality needed, i.e., editors, compilers, helpers or any software that is used within a development environment.

The plugins are the backbone of the Eclipse platform. The plugins are modules that add functionality to the platform and anyone can develop and contribute with their own plugins. As the Eclipse platform is open-source it has a freely available API that can be used to develop plugins. In addition, Eclipse offer its own Plugin Development Environment (PDE) [31, 19]. PDE is a plugin development architecture that can aid the developer in creating plugins. PDE offers ways of creating many different plugins, e.g., view based plugins (create new views), action based plugins (add menus, menu items and toolbars) and preference based plugins (custom preference page to store user related or application related data).

Eclipse comes pre-configured with a number of plugins; most essential ones are probably the IDEs for Java and C/C++. One can choose to download the Eclipse in several pre-configured versions with plugins that are useful for a variety of environments.

6.2 The consistency checker plugin

This plugin adds a view that is (by default) located in the lower region of the Eclipse window showing the details and results of the consistency check routine and also provides functionality for letting the user add and remove alignments, see figure 6.1 on the following page.

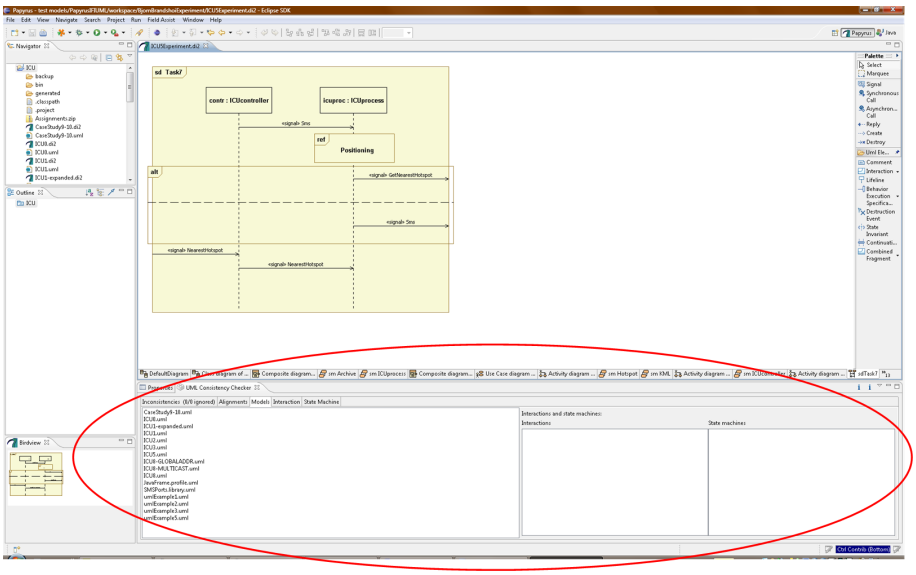

Figure 6.1: The Eclipse/Papyrus workbench with the Consistency Checker view

In addition to this plugin, other types of tooling needs to be discussed. The main type of tools we need to relate to are:

1. UML model editors
2. UML diagram editors

The plugin searches the Eclipse workspace for domain model files with the standard *.uml* extension. These files are XML files that use the Eclipse implemented UML namespace binding ([16]). The files are then parsed by the tool and the models found are added to a list of UML models for that are available to the user when creating new alignments. These files are called *domain model* files and contains the UML models. They are created by some of the available UML model editors, such as Papyrus UML [46] or Eclipse UML2Tools [16].

This thesis uses Papyrus UML, which is made in collaboration between IFI[2]/SINTEF[3] and a French team sponsored by the French Atomic Energy Commission (CEA), as the main tool to collaborate with. Papyrus UML is an open source plugin for the Eclipse platform and has its own diagram editors that work in conjunction with the Eclipse implementation of the UML metamodel[4], found in the project Eclipse UML2. The Eclipse UML2 project has implemented the UML 2.x metamodel to support the development of

[2] http://www.ifi.uio.no
[3] www.sintef.no
[4] org.eclipse.uml2.uml

modelling tools. This means that the consistency checker plugin works in conjunction with the following tools:

1. (Heavily coupled) An implementation of the UML 2.x metamodel: Eclipse UML2 [14]

2. (Loosely coupled) An UML diagram editor: Papyrus 1.9 [46]

One main goal when developing this plugin was to be very loosely coupled with dependencies. When the plugin is very loosely coupled with Papyrus, it means it will work even when there is no version of Papyrus installed. As the Eclipse UML2 package is the core of UML tool development this plugin will rely heavily on that package and it will not work without it.

Papyrus is a collection of editors for each UML diagram type (class, activity, interaction etc.) that ease the work of modelling as opposed to be working textually with the models via, e.g., the Eclipse UML2 model editor. Eclipse also has a toolbox of diagram editors called Eclipse UML2 Tools [16]. Both make use of the Eclipse implementation of the UML metamodel and the models are thus equivalent and the consistency checking plugin can read a model made by any of the diagram editors.

Papyrus generates a graphical notation file in addition to the domain model file. This graphical notation is based on an implementation of the OMG Diagram Interchange standard (DI) [40] while the Eclipse UML2 Tools uses their own graphical notation. The consistency plugin does not read any diagram files as the use of these files are not needed to do the consistency check or to show visually feedback to the users as this is done programmatically at runtime without actually changing the diagram file.

The discussion regarding differences and similarities between these two graphical notations is not the scope of this thesis.

See A on page 98 for details on the tool developed and 7 on page 65 for details on the experiment.

6.2.1 Pragmatics - functionality

When creating a tool such as the consistency checker plugin made in this thesis, one may encounter the possibility of having a tool that looks perfect on the paper that simply does not offer a practical usage for the developer using the tool. To avoid this, we need to consider this possibility and take steps to avoid ending up with a tool that the user will not use due to the lack of usability.

The ultimate goal is to create a tool that does not need the users attention at all, a tool that runs in the background and only interrupts the user when strictly needed. As this is a unicorn scenario, it is not likely to be achieved.

In this section, we investigate the main functions of the tool and looks into the possibility of achieving the ultimate goal. In the next section we

look at the practical usage of the tool when using its graphical user interface (GUI).

The main tasks for the consistency check tool are:

- Alignment of lifeline and state machine
- Consistency checking
- If inconsistencies are found:
  - Analyse inconsistencies
  - Handle inconsistencies

Alignment

The alignment of a lifeline and a state machine is a task that needs to be done manually by the user as a computer would not know what behaviour is modelled and where it aligns. Some kind of estimating or predicting could be achieved, however, by letting the user flag a lifeline with a corresponding flag on a state in a state machine while modelling, as opposed to creating the alignment after the modelling is done. Such meta statements can be found in, e.g., SPIN [23]. The algorithm would then need to parse the model and find corresponding flags to create the alignments. This way, the users would not need to explicitly create an alignment for the consistency check algorithm to run.

The easiest way to implement the creation of alignment is to simply let the user choose the elements of the alignment presented in appropriate lists. This way poses less trouble than the former proposal, at least when the user wants to add several alignments to the same model as one then would have to address the flags to a certain alignment.

To create an alignment, open the tab "Models", then choose what model to create the alignment in and interaction + state machine, see figure 6.2 on the next page. Then go to the next tab, "Interaction", and choose what lifeline in the interaction to look at, see figure 6.3 on the following page. The last step is to choose what vertex within the state machine to start looking at, see figure 6.4 on the next page. Now go to tab "Alignments" and click the "Save" button in order to save the alignment for consistency checking, see figure 6.5 on the following page.

Consistency checking

The consistency checking is an algorithm that, for the most of the time, does not need any user interaction. The algorithm runs in the background and does not prompt the user for any input unless there are certain ambiguities within the model. The consistency checker automatically starts when an

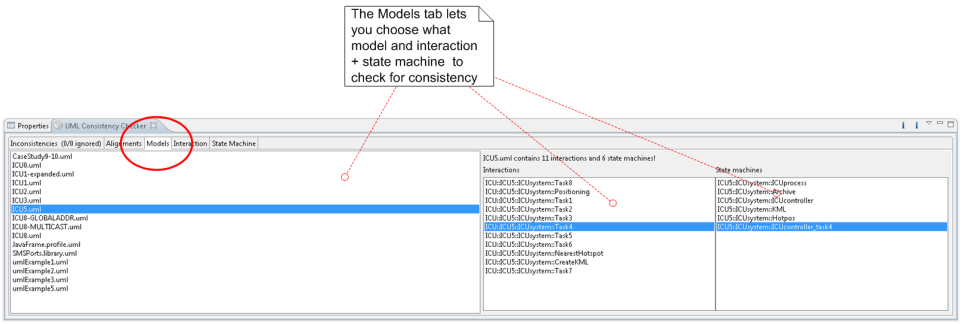

Figure 6.2: Adding an alignment - choosing model, interaction and state machine

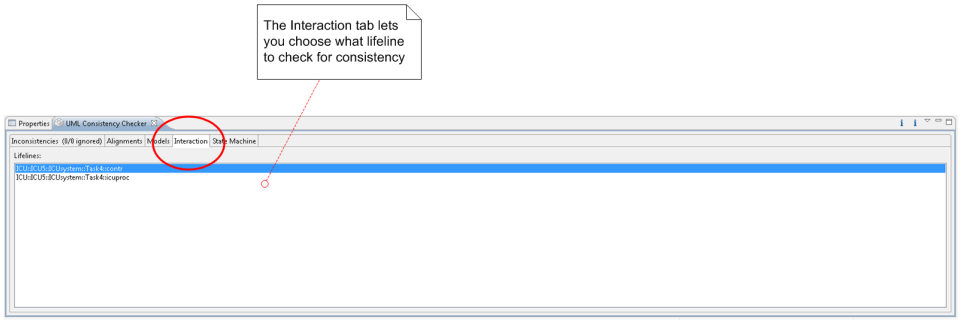

Figure 6.3: Adding an alignment - choosing lifeline

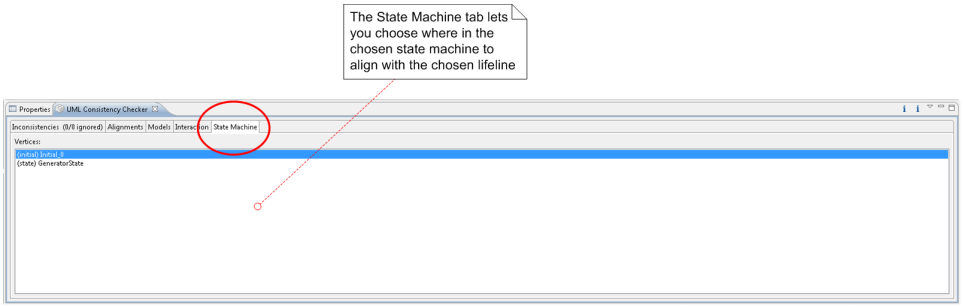

Figure 6.4: Adding an alignment - choosing vertex

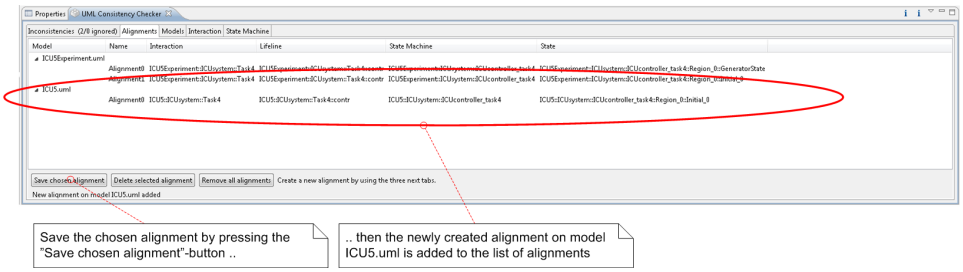

Figure 6.5: Adding an alignment - Saving and reviewing

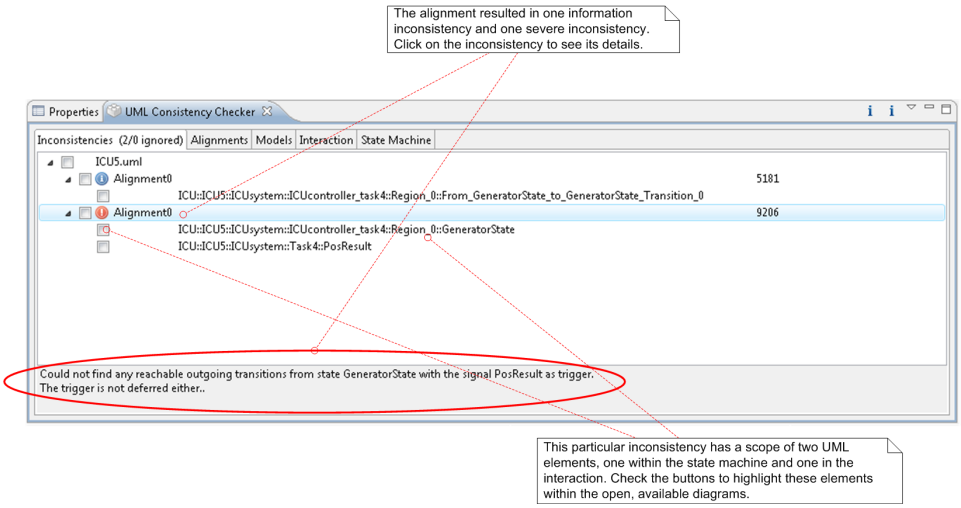

Figure 6.6: The results of the consistency checking

alignment is created and will rerun the checks that is created for a model when changes are made to it.

To see the results from the chosen alignments' consistency check, go to the "Inconsistencies" tab, see figure 6.6.

Analyse inconsistencies

If inconsistencies are found, ultimately, the tool would analyse and handle the inconsistencies without interrupting the user. In section 3.4.4 on page 18 we derive that to decide how to handle an inconsistency we need to interact with the user, making choices available together with their expected result.

There is a need for human intuition and reasoning to be able to choose how to handle the inconsistencies. If the computer was to do this analysis alone, it would need some set of rules to be able to choose an action, i.e., choose the action (to handle the inconsistency) that results in the fewest number of new inconsistencies (the action with the least number of ripple effects).

Handle inconsistencies

Handling the inconsistencies could be done automatically by the tool by, e.g., adding an expected trigger to a transition. This action is also possible to do manually by the user, but when an action to handle an inconsistency is chosen in the analysis, the changes that needs to be done to the model is unambiguous and could therefore be done automatically by the tool.

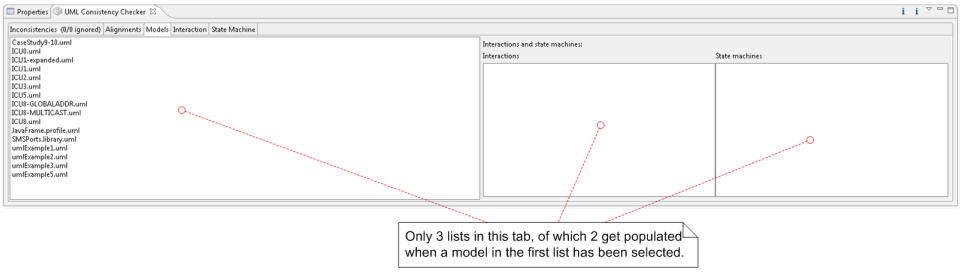

Figure 6.7: GUI with little information to it lets the user concentrate on the few elements rather than getting annoyed of the many.

6.2.2 Pragmatics - GUI

Creating a good and intuitive Graphical User Interface (GUI) for a program is of outermost importance. The key is to design a GUI in such way that it bridges the users expectations of the tool's look and feel. As most software engineers are not designers (and vice versa) this poses a problem when trying to develop a killer application that not only works well behind the scenes, but is also usable. This section will discuss some of the principles of good GUI design and relate these to the tool being developed[5].

As the developer is working on an application, it is easy to create a GUI that suites the developer more than the user. As the user is a novice in using the tool and does not know everything that the developer knows, the GUI must be made clearly and intuitive even for the first-timer. A user that is presented with a non-intuitive GUI quickly bores and give up. We have tried to follow these guidelines by letting the plugin consist of several tabs with little information on them, see figure 6.7. This way, hopefully there will be no "information overload" and the user will clearly see what tab to use when doing things.

Letting the user have full control of the GUI (or at least give the user the "impression" of having full control) is important as well. The user can quickly become irritated and annoyed if presented with a lot of functionality that is not available due to some (badly designed) constraint that is not met. The application should give the user fully control to navigate and use its functionality in an ad hoc order. This, of course, implies that the application must be designed in a way that it can actually cope with these conditions. If the application is designed to work in a sequential way, then the GUI design must visually render this in a way that the user still feels like being in control, i.e., by making options transparent when they are not available instead of having them grayed out.

The plugin consist of tabs that actually need some form of sequential activation. For instance, when creating the alignment, the user must use the

[5] Inspired by [29]

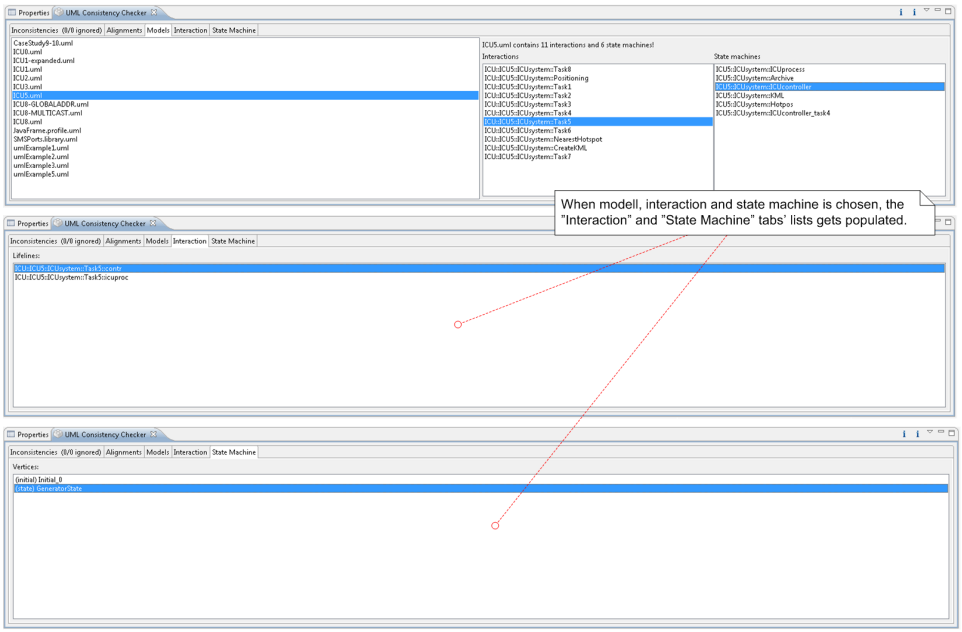

Figure 6.8: GUI with no physical restrictions. Lists gets populated dynamically.

three tabs designed for that and then go to the "Alignments" tab to save it. The three tabs for creating an alignment is activated by each other, i.e., when a model is chosen the list of interactions and state machines within it gets populated. This guides the user through the process without taking the feel of control away, see figure 6.8.

A novice user should not be presented with a top level GUI that is overwhelmed with functionality. Less is more and it clearly makes it simpler to understand the GUI if it is designed such that the user input is split into an iterative fashion other than presenting everything at the initial screen. These former design principles are all basics for a good, intuitive GUI design.

Further, all good GUI designs share many common characteristics, like the use of real-world metaphors whenever possible. This means that instead of using a textual label saying "Warning", one can use a picture of a **yellow** exclamation mark. Likewise, instead of having a text saying "Error", one can use a picture of a **red** exclamation mark. These pictorial representations are easier for the human mind to process and thus makes the whole GUI experience easier to work with. Icons must be used with caution though, as different people has different perspectives on things. While the developer find it perfectly intuitive to use a certain icon, a regular user may have no idea what the icon resembles. The use of icons is best when they represents well known everyday metaphors that "everyone" find intuitive. See figure 6.9 on the next page

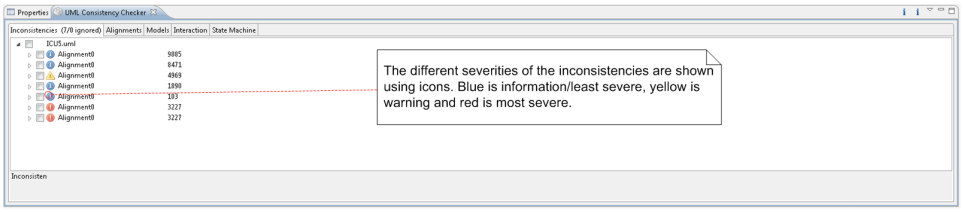

Figure 6.9: Icons represent the severity of each inconsistency. Blue is least severe, red is critical.

It is also important to maintain consistency in the use of words and terms in the GUI. If the user is presented with the use of the term "interaction" in one place, the GUI should stick to that term whenever addressing the same concept. If the GUI suddenly used "sequence diagram" and "interaction" interchangeable, it might lead to confusion and frustration amongst the users. In addition to keep the terms consistent, all text that is presented to the user should be short and concise. This applies to graphical elements, such as buttons and labels, but also in error messages and other kinds of textual feedback.

The Consistency Checker tool strives to adopt these principles, read more in section A.2 on page 100.

6.3 An iterative development

To start with, the plugin was integrated into the Eclipse workbench as a simple view where the user got feedbacks from the check routine to the console. The user had to implicitly create the alignment and start the routine each time changes were made to the model. Due to this severe restriction in user-friendliness, in the next iteration the plugin stored the alignment from each run, so that the user would not have to create the same alignment several times. Also, the plugin did automatically update the consistency check routine (by running it all over again) whenever changes were made to the model. This improved the user friendliness a lot and the plugin could run in the background without interrupting the developer during changes of the models.

Still, the plugin only had textual feedback via the inconsistencies tab of the GUI (see section A.2.1 on page 101) which could make it difficult to actually locate the scope of an inconsistency within a given diagram.

In the next iteration, the plugin got loosely coupled with the Papyrus tool and showed the scope of an inconsistency visually in the diagrams by highlighting elements of an inconsistency's scope. This helps the developer a lot as there can be a lot of models and the qualifying name of model elements are not as easily read.

The plugin still does not not take actions or make any analysis if inconsistencies are found, it only reports its findings to the developer, letting the developer choose to try to deal with the inconsistencies or not.

Ultimately, the plugin will run instant consistency checking and analysis in the background while the developers are working on their models. In addition it could be possible for the plugin to act, trying to fix intolerable inconsistencies or at least give the developers hints on what to do to improve their specifications by predicting the outcome of actions.

No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

Albert Einstein

There is no such thing as a failed experiment, only experiments with unexpected outcomes.

Richard Buckminster Fuller

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger Dijkstra

7 Experiment

The work of this thesis resulted in a developed tool for the consistency checking method of interactions and state machines developed. We ran an experiment on the tool, which was aimed to test the *pragmatics* of it. The participants of the experiment was a carefully selected group of persons which all are natural end-users of such a tool in question. All of the participants are computer science students from the Department of Informatics, University of Oslo. They are all attending to a course in modelling[1] and are a typical targeted group for such a tool. There were in total 13 participants which attended to this experiment, ranging from bachelor- to phD students and all had some prior knowledge of modelling and UML, but most of them were unfamiliar with the notion of checking models for consistency.

The participants did ten modelling assignments while interchangeably using the tool as an aid in the work and then at the end graded and evaluated the tool answering a questionnaire. The experiment was done in an closed environment for two hours and the participants did not have access to the assignments prior to the exercise, but they did have the opportunity to test the tool for three days in advance.

The assignments were tested on a selected minor group of equally typical users prior to the actual day of the experiment. This was done to see whether the degree of complexity of the assignments were satisfactory taken the participants prior knowledge of the domain into account. When we tested the assignments, we specifically noted down the time each assignment took in order to be able to estimate how many assignments to create for the session. It showed that the average time for each assignment were around five min-

[1] The course is INF5150 at the Department of Informatics, University of Oslo.

utes and thus we made ten assignments in total for the actual experiment as we had an hour exactly for solving the assignments.

7.1 Experiment overview

As time were of the essence when doing this experiment, it was carried out as a pilot study rather than a full-fledged empirical study. The main goal of the experiment was to assess the tool developed, observing the usability and getting feedback from the users but also to evaluate the concepts of consistency checking as developed in this thesis. To be able to run such a pilot study, we had to take the participants into account and making the study a two-way street, i.e., making it interesting and valuable for both parties [8]. As the participants were attending a course where some of the objectives are to learn the usage of sequence diagrams and state machines, we tried to make the study closely attached to the curriculum by creating the assignments based on their current work.

By doing this, the students got to work with relevant assignments while validating the research hypothesis. These pedagogical considerations hopefully gave the students inspiration to attain the study while giving us valuable feedback on the work of this master thesis. In addition, there was a small prize for every attending participant and a main prize for the best solved set of assignments.

7.1.1 Goals

To establish goals for the experiment, we created some hypotheses that needed evaluation:

- **The concepts of consistency checking are easy to understand.**
- **The consistency checking tool is intuitively easy to use.**
- **The consistency checking tool helps the developer in keeping the specifications consistent.**
- **The consistency checking tool makes the developer more efficient.**

These hypotheses are not completely independent, e.g., the tool will probably not be used by many developers if its usability is poor, even if its functionality is good. This means that the tool might have well designed interior but as long as the exterior is badly designed, few developers will endure using it. Also, the tool might have both well designed functionality and graphical user interface but if it does not give the developer the possibility of becoming more efficient in her work, it will probably not be used then either.

Therefore, the hypotheses are equally important means as to evaluate both the method and the tool developed in this thesis.

7.1.2 Preparation

The experiment was carried out on a Monday and was introduced to the students on the prior Friday. As we only had the participants available for 90 minutes, we made a short introduction prior to this to make sure that they knew something about what was going to happen and also to make the tool with its documentation available for a preview during the weekend.

On the actual day of the experiment, the participants were briefed on how the experiment was to be carried out (15 minutes) before the tool (for those who had not downloaded it prior to the session) and assignments were distributed amongst them (60 minutes). The last 15 minutes of the session was used to fill out the questionnaire to evaluate the usability of the tool and the concepts of consistency checking.

The assignments included pre-made models with specifications made with sequence diagrams and state machines, see B on page 112 for details and section 7.1.3 for an example of a typical assignment that was given. The participants also got printed hand-outs with the models to write down the alignment and changes needed to be done to make the specification consistent along with the time spent on each assignment.

As one of the main concept of the consistency checking routine is the alignment of an interaction and a state machine, this was where the assignments started. All of the models had minor changes done to them so that some were consistent at the time they were handed out and some were not. The job was to align the specifications and then checking them for consistency and writing down how to solve the inconsistencies found, both with and without using the consistency checking tool.

The participants was divided into two groups, all working on the assignments individually. The first group did one half of the assignments using the tool as an aid and the other group did the second half of the assignments using the tool.

The questionnaire gave each participant the possibility to evaluate their use of the consistency check tool and the tool it self. See C on page 137 for details on the questionnaire.

7.1.3 Example assignment

An example assignment was given by an interaction (see figure 7.1 on the next page) and a set of state machine (one for each lifeline) (see figure 7.2 on the following page and 7.3 on page 69) and the task was to align the lifelines an state machines and check for consistency.

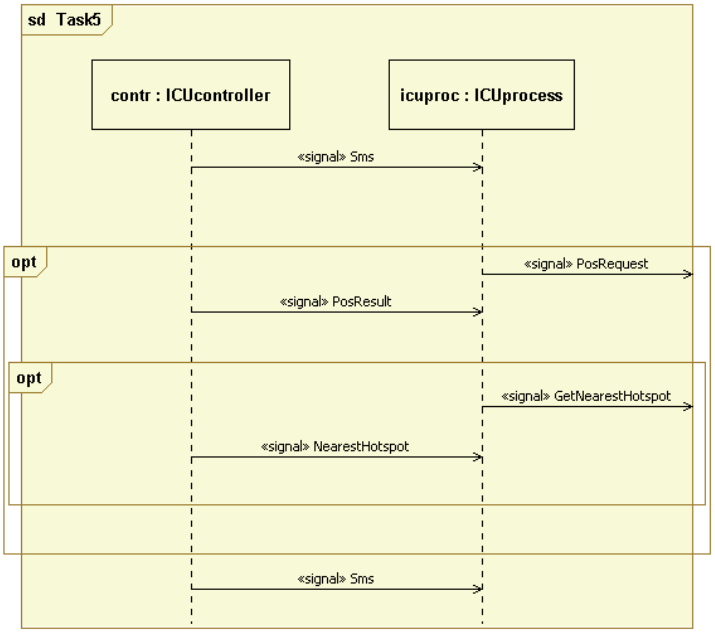

Figure 7.1: Example assignment - Interaction

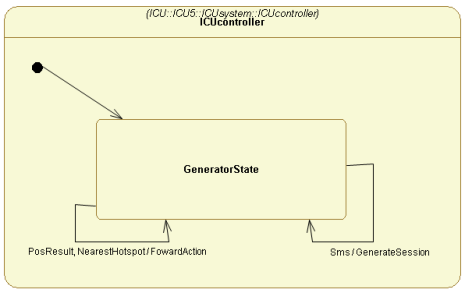

Figure 7.2: Example assignment - State machine ICUcontroller

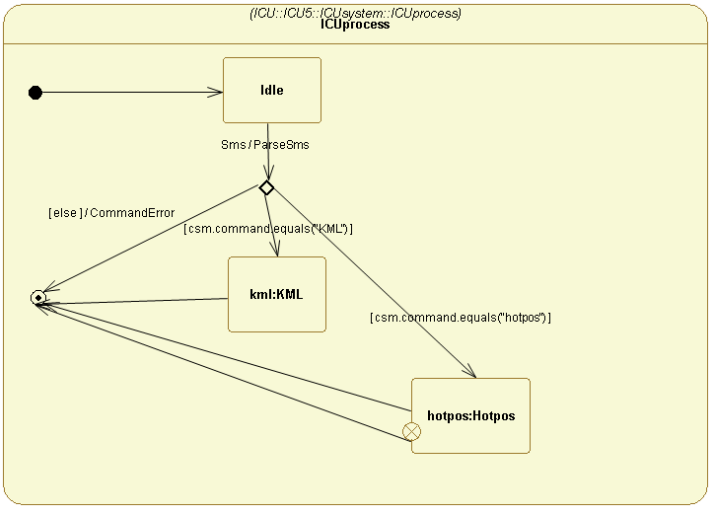

Figure 7.3: Example assignment - State machine ICUProcess

7.1.4 Running the experiment

We had 60 minutes for solving the assignment, making the average expected time for solving each assignment approximately 6 minutes.

For each assignment, this was the task sequence:

1. Write down the start time on paper
2. Figure out what lifeline the state machine represents
3. Align lifeline and state machine
4. Write down the alignment on paper
5. Make the specifications consistent (manually/tool) by making changes to either one if needed
6. Write down the changes on paper
7. Write down other possible ambiguities or errors
8. Write down the end time on paper

When the time for solving assignments had expired, the questionnaire was handed out to rate their experience they had gained using the method and tool.

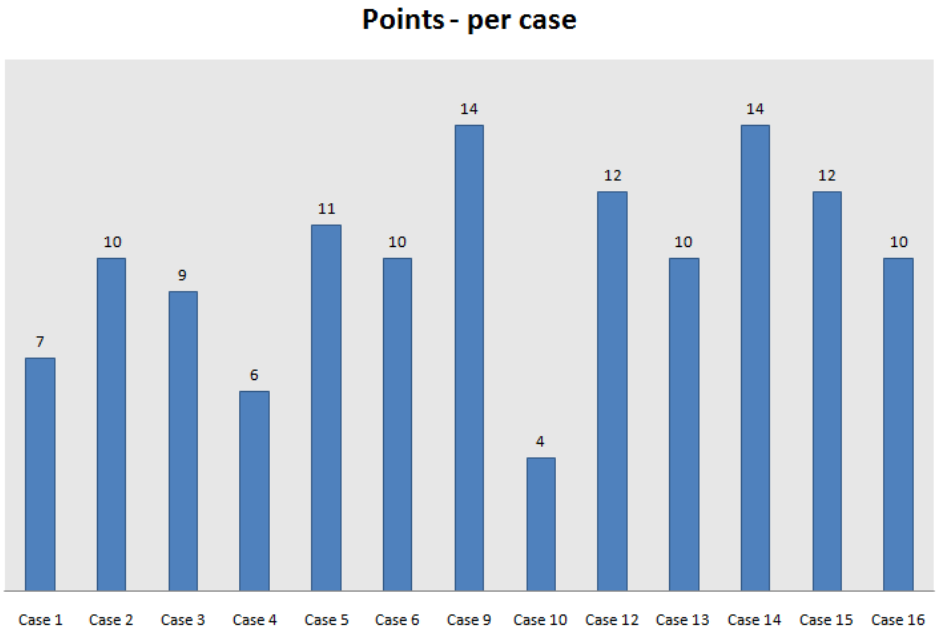

Figure 7.4: Assignments - Score for each participant

7.2 Experiment results

When the experiment session ended, we had the results from each assignment made by each participant which were evaluated. We could now compare the tool-solved assignments versus the manual-solved assignment to see whether the tool-solved ones were solved faster and more correctly. In addition, we had the subjective evaluations for each participants from the questionnaire that helped us determine whether the tool is easy to use or not and if the concepts in this domain is reasonable easy to understand.

We were interested verifying the hypotheses presented in 7.1.1 on page 66.

When this tool was made available no documentation, other than a small quick-guide within the plugin, were available. By having more proper documentation available for the tool, the results from the assignments and questionnaire could have been further improved as this would allow the participants to become better at using the tool prior to the experiment session.

See the results in detail for each question in the questionnaire in section C.1 on page 137. Each participant had their result subjectively graded, which is shown in table 7.4, which favors case 9 and 14 with the best results. As case 14 was anonymously delivered, case 9 won the main prize.

7.2.1 Interpreting the results

The experiment suffered from an unexpected turn, which disfavored the results, during the installation of the assignment models for $\approx$ 50% of the participants. Unfortunately, it showed that these participants did not create the appropriate project type within the Eclipse workbench at the start of the experiment. This might have been caused by the lack of proper information prior to the session starting, although all participants had learned that in order to work with the models used in the corresponding course, they would have to create a new "JavaFrame project" and not the regular, simple "General project" in Eclipse.

By creating a JavaFrame project, two libraries are added to the project automatically which includes signals used in the assignments during the experiment. For those participants who did not have these libraries, the tool unfortunately did not give any feedback when it should have, as it failed and threw an exception in the background. The participant then mistakenly interpreted the empty feedback as a positive feedback, i.e., the specifications being consistent - which in 7 out of 10 assignments they were not.

This shows the importance of having a tool giving feedback to its user in any case. Even for a tool like the Consistency Checker plugin that is checking for erroneous specifications and the users are expecting feedback when errors are found, it is still important to give feedback even when there are no errors to report.

It also shows the fragile state of running an experiment which deals with many participants and their computers, as there are numerous things that can go wrong. Even though this experiment was precisely planned, the participants were carefully selected and all assignments were tested beforehand both manually and by using the tool, it shows that there were still factors that were unattended and which caused some of the assignment answers to be faulty.

The consequences were that $\approx$ 50% of the tool-based solutions are unusable. This resulted in interesting results which can seen in figure 7.5 on the next page and figure 7.6 on page 73. Assignments 4-7 had a higher correctness when solved manually and the tool-based solutions had decreasing correctness from assignment 4 and onwards, this was very surprising as we expected the tool-based solutions to be increasingly more correct than the manual solved ones as the complexity of the assignments escalated increasingly. We also expected that virtually all tool-based solutions was correct.

The correctness of these tool-based assignments would have been close to 100% with the tools' environment accurately set up.

It should be noted that inconsistencies were present in assignment 4-10. The tool-based solutions for assignment 1-3 were probably also affected by this problem, but as the tool did not give any feedback for consistent specifications, the answers were correct even so.

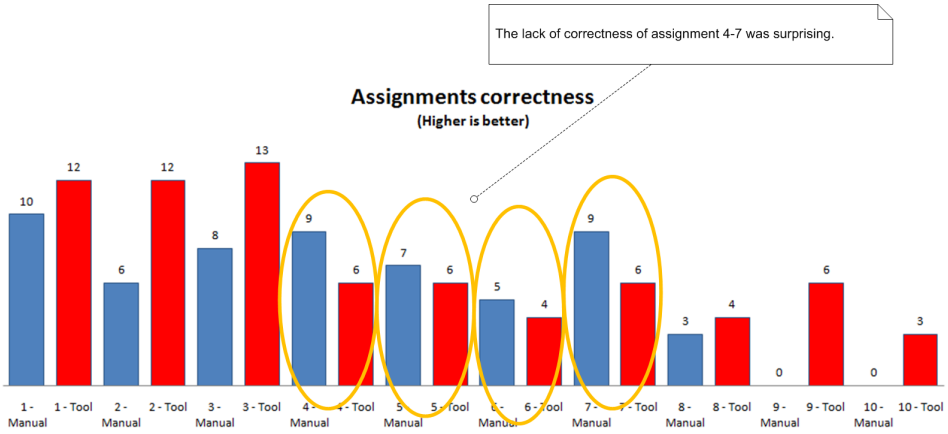

Figure 7.5: Assignments - Correctness of each assignment, commented

This is what led us to the investigation that led to the discovery of the missing libraries for certain participants.

Even so, the overall results from the experiment showed valuable tendencies which are interpreted and discussed in the next sections.

7.2.2 Hypothesis 1: Concepts

Whether the user fully understands the concepts of consistency checking as presented in this thesis is critical for the further use of the tool and method. The user needs to be comfortable with the concepts of how to check the specifications for consistency. This includes how to align the lifeline with the state machine and what to look for. The two first questions of the questionnaire are relevant for this, see section C on page 137.

The majority of the participants found the concepts of consistency checking above average easy to understand. The graph (see figure 7.7 on the following page) is almost equally distributed which indicates that the concept needs some introductory lessons before working with it, but it also indicates that it is reasonable to understand. This is what we expected and hoped for as both the result of having the concept too easy to understand and the result of having the concept way too hard to understand would have been much worse to cope with. One of the comments made by the participants were: "The concept is easy but should be precisly defined". This could indicate that we either did not present the concept as good as we hoped for or it could mean that the concept could need some more formalized way of definition.

We conclude that any developer with some prior knowledge of modelling with UML would fairly easy understand the concept of consistency checking

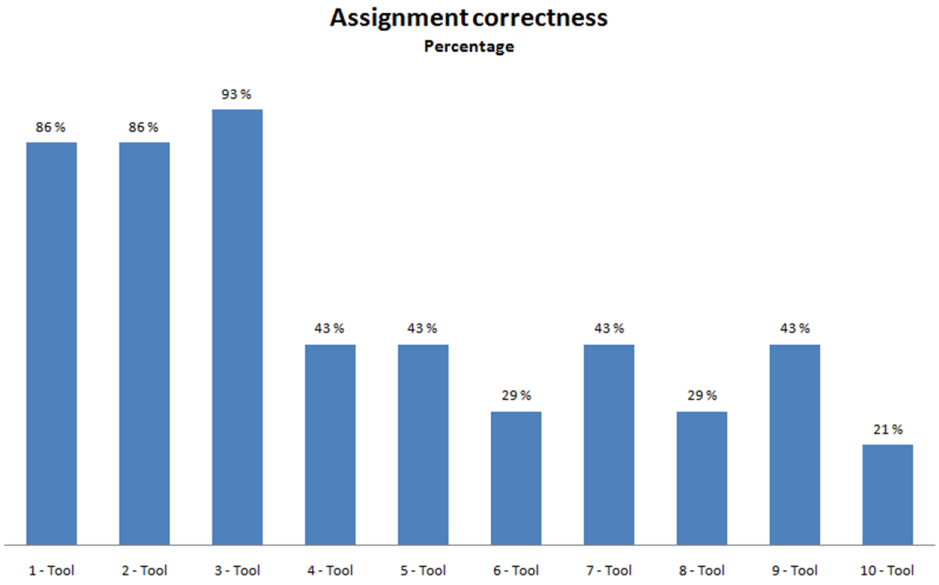

Figure 7.6: Assignments - Correctness of tool-based solutions

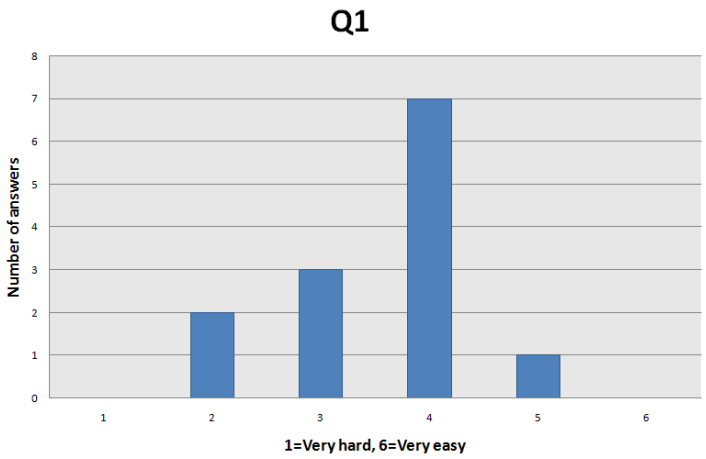

Figure 7.7: Q1: How easy is it to understand the concept of consistency checking a lifeline and a state machine?

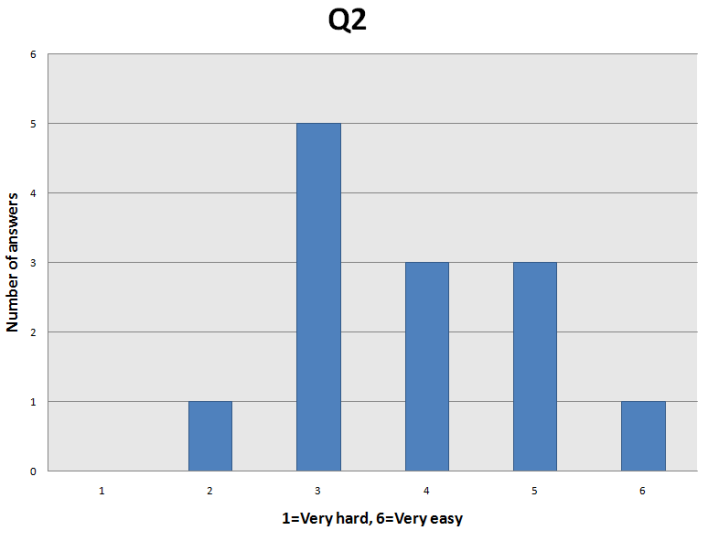

Figure 7.8: Q2: How easy is it to understand the concept of the alignment of a lifeline and a state machine?

presented in this thesis.

The majority of the participants found the concepts of alignment of a lifeline and a state machine above average easy to understand. The graph (see figure 7.8) shows that the weight is distributed along the grade of 4-5 which is in favor of a positive result (in the sense that the easier to understand the concept, the more positive is the result). One of the comments made by the participants were: "In the experiment it was easy to align because of naming but generally can be difficult because one should look at behaviors.". This can indicate that the process of aligning a lifeline with a state machine can be a bit tedious when selecting the state, as one needs to know what behaviour to look for within the state machine to make this choice.

We conclude that the concepts of aligning a lifeline with a state machine is above average easy to understand.

7.2.3 Hypothesis 2: Ease of use

To evaluate whether the tool is intuitively easy to use, we needed to make use of the data from the final questionnaire, see section C on page 137. The reason for this is whether this hypothesis is verified or not is only possible to answer when the tool has been properly tested and evaluated by the users. As the questionnaire was answered after doing the assignments, the participants had by know gotten knowledge about both the tool and the method. Question 3, 4 and 5 are of interest to help verify this hypothesis.

The majority of the participants found the graphical user interface easy to understand. The graph (see figure 7.9 on the next page) shows that the

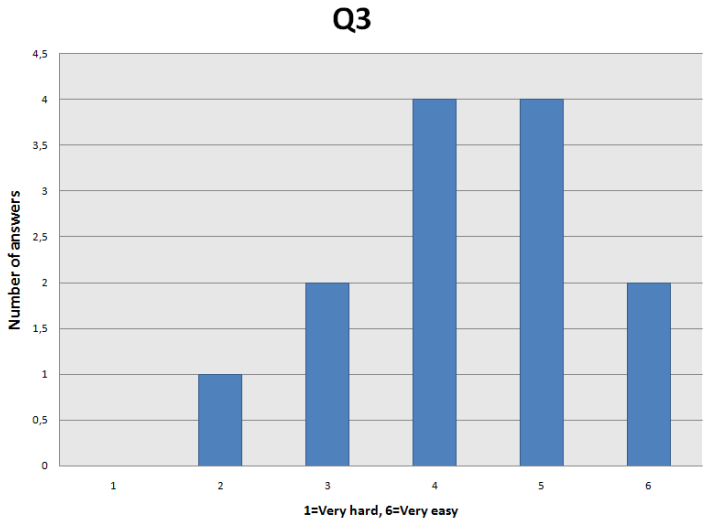

Figure 7.9: Q3: How easy is it to understand the graphical user interface (GUI) of the consistency checking view?

weight of the graph is distributed along the grade of 4-5 which is in favor of the GUI to be easy to understand. Although some felt the GUI as a bit tedious, the results were satisfactory and goes a long way to verify the hypothesis.

The vast majority of the participants found the functionality of adding a new alignment in the consistency view easy. The graph (see figure 7.10 on the following page) shows that only two participants have graded the question to a grade 3 and the rest to grade 5 or 6. This indicates that to add an alignment is easy, although some of the comments made indicates that there is improvement to be made to make it even better (see list of comments below).

The participants were quite divided when it came to grading how easy it is to interpret the feedback/results from the plugin (see figure 7.11 on the next page). The vast majority of the participants graded it to a 3 or a 2, while four participants graded it to a 5 or 6. This can indicate that you have to understand the concepts of consistency checking before the results from the tool makes any sense. It could also indicate that they way the results are presented needs some refinement, like making the language used easier to understand and increase the use of graphical notations.

We conclude that the GUI in general and how to create an alignment is easy to understand, but the results/feedback given by the tool needs to be refined.

These are the comments made to the GUI-related questions given by the participants:

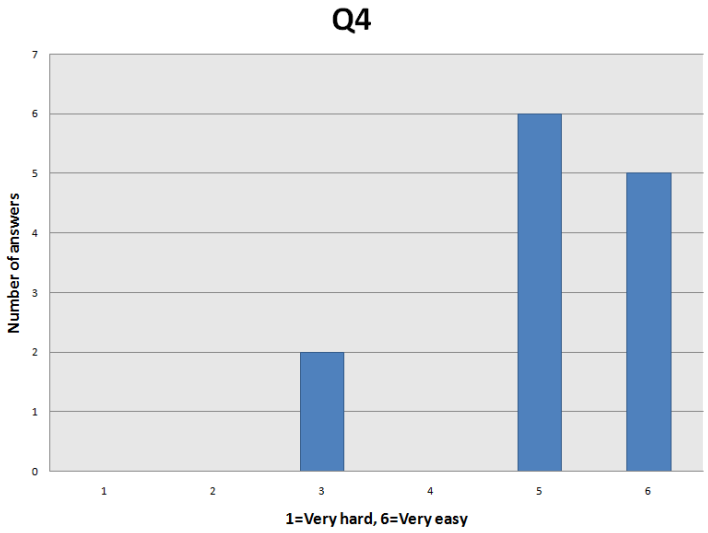

Figure 7.10: Q4: How easy is it to add a new alignment in the consistency checking view?

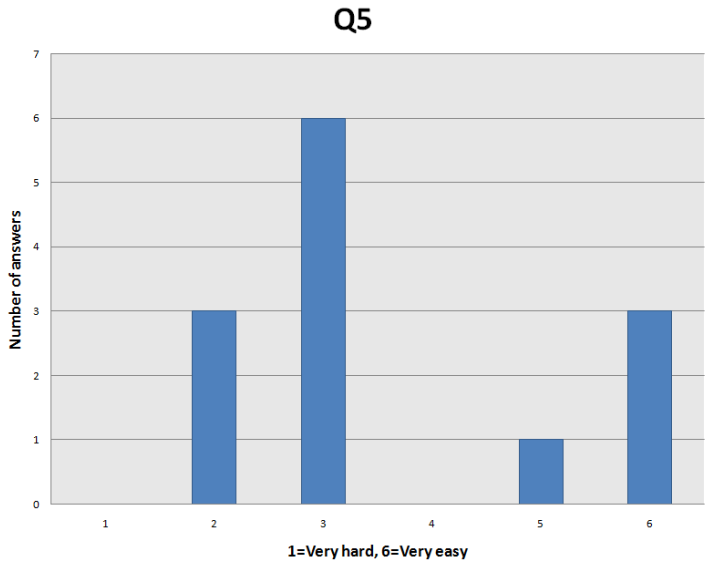

Figure 7.11: Q5: How easy is it to interpret the response/results from the plugin?

- *GUI should include a wizard.*
- *GUI should present steps in a logical order.*
- *Would do better with better UI.*
- *Some results were not very understandable, it may help to have better log (..).*
- *Just like compilers, one needs to be used to interpreting the feedback messages.*
- *I would change the order of tabs, first model, interaction&state machine, alignments and then the results.*
- *The tool is easy to use, the results which it gives explain good the reasons of inconsistency.*
- *The use of the tool is not that intuitive at the first place; More relevant information should be provided.*
- *It worked very well after I first understood how to use it. A very good tool!*
- *Easy to add a new alignment when you know how.*
- *The list of models should include path for each model as there can be many models with the same name in the workspace.*

They all indicate that the GUI can be made better with some reorganizing of the GUIelements and perhaps implement some sort of helper (e.g., a wizard), to get the user started.

7.2.4 Hypothesis 3: Helps keeping specifications consistent

This hypothesis was designed to evaluate whether the tool helps the developer in her modelling process or if it simply adds more work process and thus making it more complicated and time consuming. This evaluation is based on the answers in the questionnaire as well as the results of the assignments. The questionnaire told us whether the user felt the use of the tool as a positive enhancement or a liability. The results of the assignments told us whether the tool-based assignments were more correct than the manual-solved ones.

Question 6-8 are of interest to help verifying this hypothesis, but also question 9 and 10 help indicate the value of the tool.

If the tool is to help the developer in her modelling environment, it needs to be reliable and trustworthy for the developer to use. The vast majority of the participants actually used the results the tool gave when

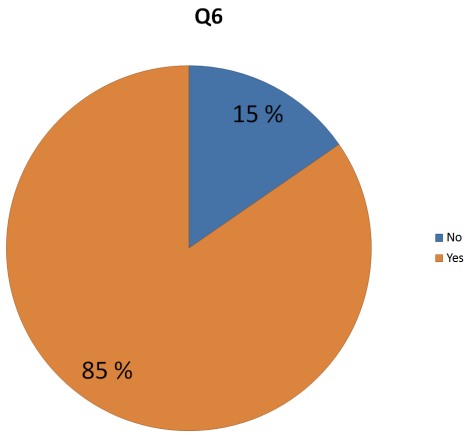

Figure 7.12: Q6: Did you use the response/result from the plugin when checking for consistency?

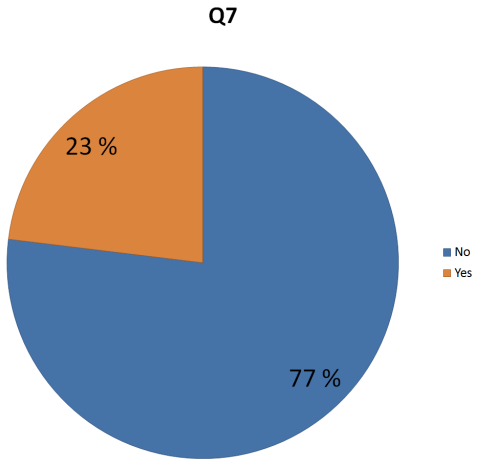

Figure 7.13: Q7: Did you trust the manual checking more than the results from the plugin?

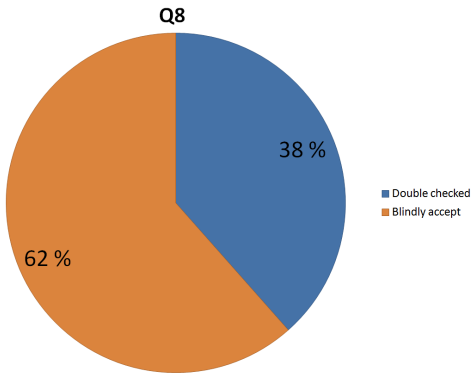

Figure 7.14: Q8: Did you *blindly accept* the results from the plugin or did you *double check* the answers manually?

checking for consistency, as the results of question 6 show (see figure 7.12 on the preceding page). This indicates that the tool does its work (i.e., consistency checking) in a way that its feedback can actually be used by the user. This helps backing up the hypothesis that the tool is indeed aiding the developer. To further verify this, the vast majority of the participants found the manual checking less trustworthy than the checking the tool performed, as the results of question 7 show (see figure 7.13 on the previous page). Although the number of participants that actually used the results from the tool in their consistency checking was higher than the quantity that trusted the results given by it. This indicates that some actually used the results from the tool even though they trusted the manual checking more, which either means they did not know how to do the checking manually and had to use the tools' result or they somehow felt obligated to use the tools' results. As this only applies to some few participants it poses no real threat to the hypothesis.

When we add the results of question 8, which asked whether the participants blindly accepted the results from the tool or they double checked the answers manually, see pie chart (figure 7.14), we see that almost 40% did actually double check the answers manually. As some of the comments (see below) indicates, they did the double checking to see whether the results from the tool were reasonable and made sense. As this was the first time the participants used the tool extensively, it could indicate that the tool needed time and usage to gain the users trust before they blindly accept its results and it show that the participants were critical and indeed interested in verifying the results from the tool which is a good thing. This indicates that the developer would like to be presented with feedback in a way that she do not feel like she has to do the double checking to be sure of it. This can again indicate the lack of trustworthy feedback from the plugin. Maybe the feedback can be re-arranged in a way that the user get to know what she

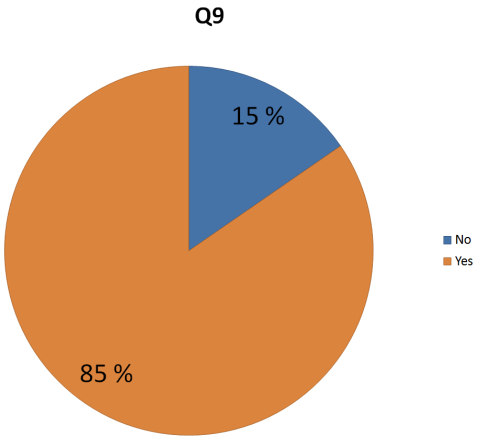

Figure 7.15: Q9: Do you think you could make use of the tool in the course?

need to be satisfied and do not feel the urge to do any double checking.

The last two questions round it off by asking the participants whether they thought they could actually use the tool further in the modelling course they were attending[2] and by letting them grade the tools' overall value. See pie chart (figure 7.15) and graph (figure 7.16 on the following page) for the results. Whether the ninth question actually has any value or if the participants felt like obliged to answer positively is not trivial to assess, as 85% of them answered that they could imagine using the tool further in the course. If we assume that the results are a matter of fact and are truthfully verdicts from the participants then it is a testimonial to its usability. This is further backed up by the last question where the vast majority of the participants grade the overall value of the tool to 4 or 5, which is above average. These last two questions are lightly taken into account when verifying this hypothesis.

Comments relevant to this section given by the participants:

- *It makes life a bit easier, since you almost do not have to think when using the tool.*
- *I found tool better an easier to use than manual consistency checking.*

To further verify this hypothesis, we evaluated the results from the assignments.

As the table and graph (7.17 on the next page and 7.18 on page 82) show, the assignments solved with using the tool were more correctly solved than the manually solved ones even though the tool-based solutions were affected by the unexpected factors mentioned in section 7.2.1 on page 71.

[2]INF5150, Department of Informatics, University of Oslo

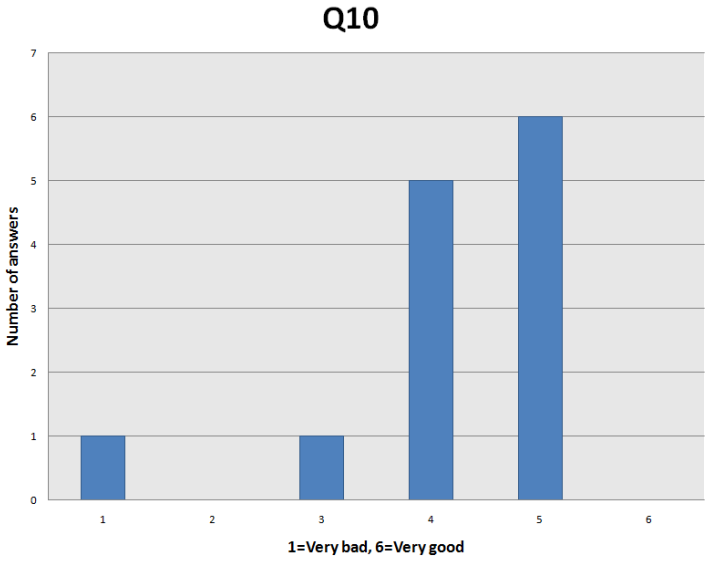

Figure 7.16: Q10: Overall value of the consistency check tool

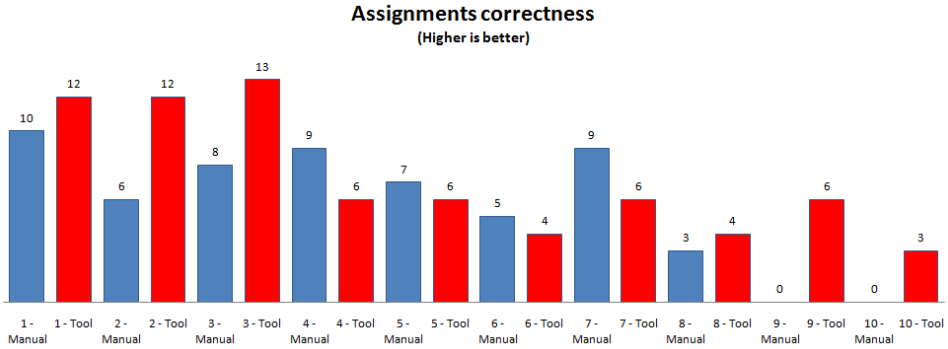

Figure 7.17: Assignments - Correctness of each assignment

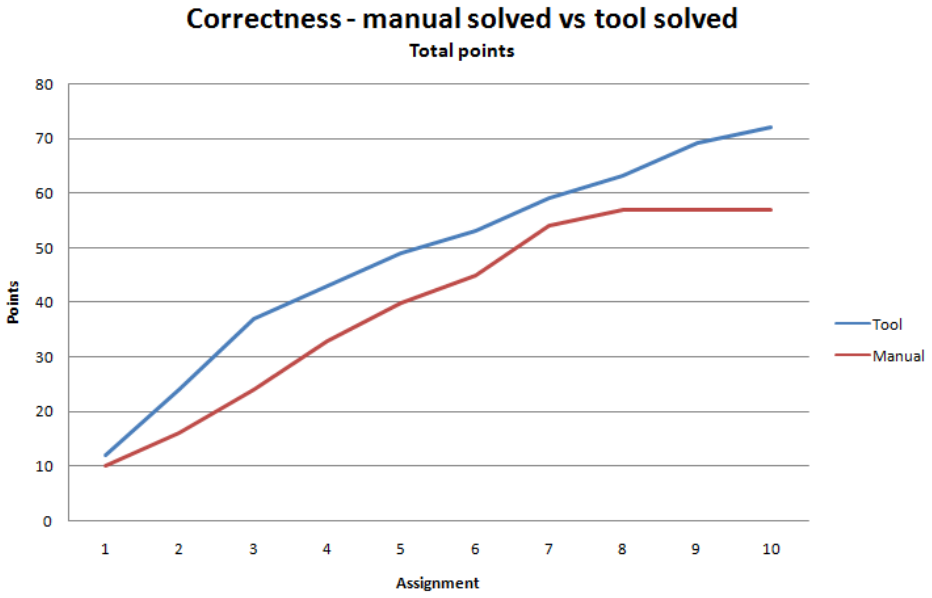

Figure 7.18: Assignments - Correctness - Manual solved vs tool solved assignments - Points

These conclusions are based on an subjective assessment that graded each of the results from the assignments with a value from 0-2, where:

0 - No real answers given

1 - Some of the inconsistencies found / some valuable answering / some understanding of the assignment shown

2 - All inconsistencies found and the participant showed an above average understanding of the assignment

But if one take a closer look at the correctness for each assginment, see figure 7.19 on the following page, one see that only 60% of the assignments solved with the tool were more correct. This is interesting and somewhat disappointing as we were expecting nearly 100% of the tool-solved assignment to be correct. This indicates that the participants did not know how to use the tool and how to interpret the feedback given by it as the fact is that the results given by the tool for these assignments, when the alignments are properly created, are correct. This also helps backing up the conclusion made in the previous section.

We conclude that the tool is doing what it is supposed to and it is trustworthy in its use and increases the models correctness, and thus helps the developer keeping her specifications consistent. However, some more work

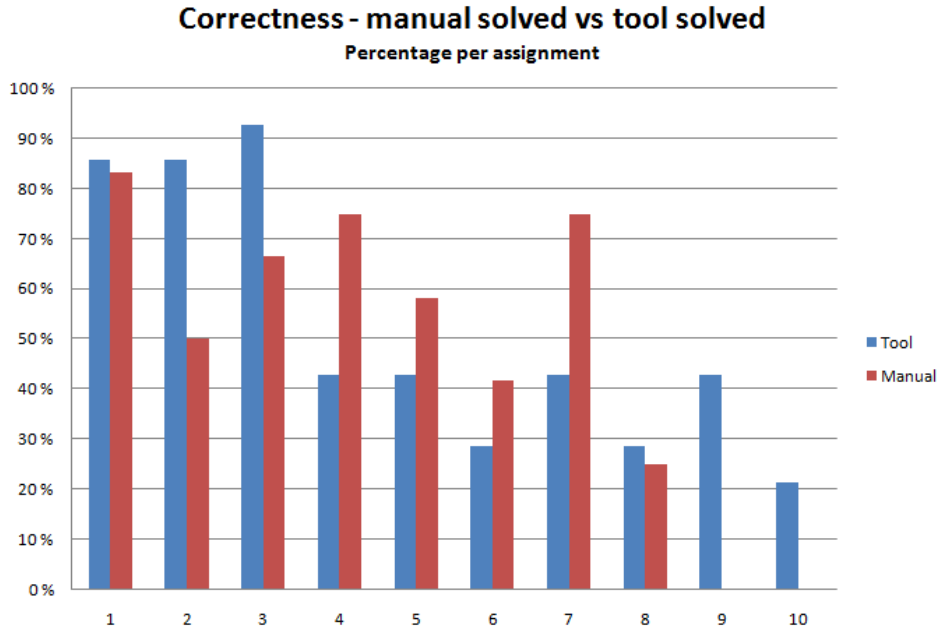

Figure 7.19: Assignments - Correctness - Manual solved vs tool solved assignments - Percentage

need to be done to convince the user that the results are accurate in order to further increase the value of the tool.

7.2.5 Hypothesis 4: Tool efficiency

Having a tool that actually does what it is supposed to do is only one of the objectives. A tool like the consistency checking tool is developed for the purpose of aiding the developer model more consistent and efficient. The hypothesis evaluated in this section is whether the tool increases the efficiency of the developer. The data collected from the results of the assignments gave us valuable data to use in which to answer this question.

The average time graph (see figure 7.20 on the next page) show the average times used per assignment for all participants. What is interesting to notice about the first assignment is that the use of tool probably made the solving more complicated and time consuming as the assignment is simple and is easy to solve by hand. The participants did this assignment faster by hand than by using the tool. This indicates that the specifications needs to be more complicated than presented in this assignment for the tool to actually be valuable in using.

The rest of the assignments, with the exception of number eight, the tool-based solutions were faster than the manually solved ones. This is the result

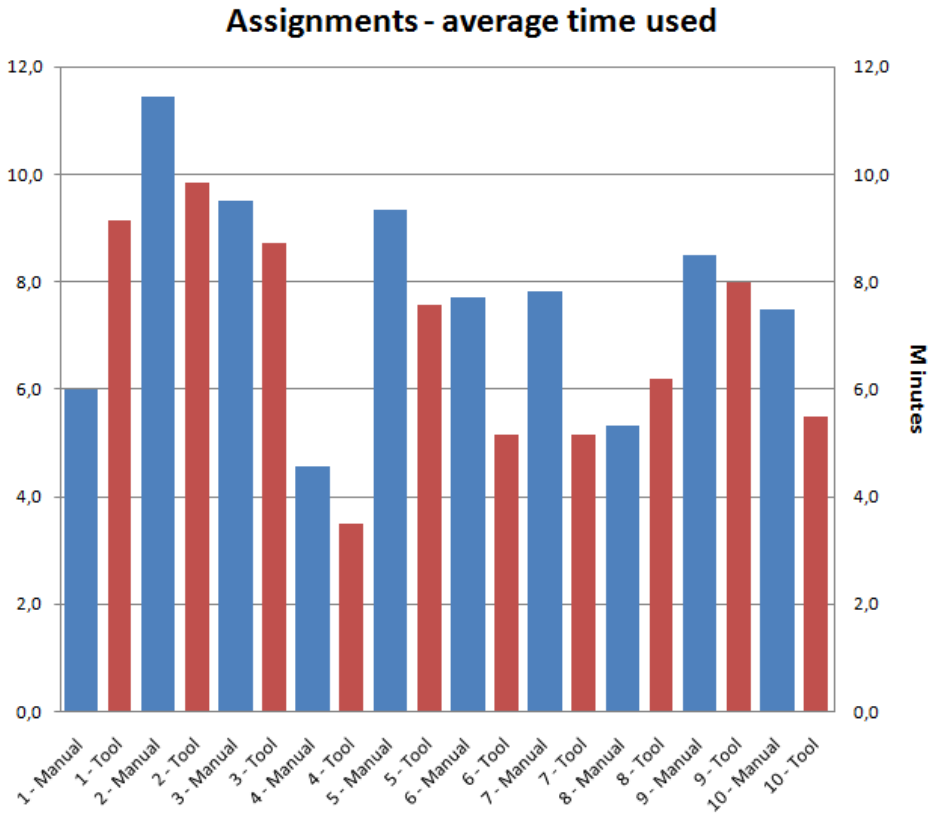

Figure 7.20: Assignments - Time used

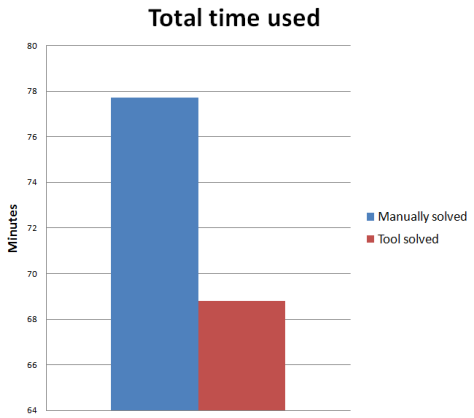

Figure 7.21: Assignments - Total time used

we hoped for and helps verifying the hypothesis evaluated in this section.

If we look at the total time used for all assignments (see figure 7.21) we clearly see that the use of the consistency checker tool improved the users efficiency, although not by as much as we hoped. As the total time for the manual solved assignments were 77,7 minutes and the tool-based ones 68,8 minutes, the overhead of doing them manually is only **13%**. Clearly, we hoped for a better result than this, but there are a number of factors to why the tool did not improve the users efficiency by more than this. First and foremost, it was quite obvious that the users spent time on trying to understand the usage of the tool. They were introduced to it only a couple of days prior to the experiment and had only gotten a short introductory lesson to the tool and most of the participants probably did not try out the tool on their own prior to the experiment. This added some overhead to using the tool that for future use will be reduced when the user gets to know the tool better and thus her efficiency with using the tool and her overall efficiency when modelling will improve likewise.

The design and contents of the assignments also were of interests to how efficiently they were solved. The easy ones are easier to solve by hand and thus the use of the tool on these assignments only added overhead to the time spent. When it comes to the more difficult assignments, where the specifications typically contains many traces, the manually solving gets exponentially harder. The more complicated the specifications get, the more there is to gain from using the tool.

We conclude that the tool serve as an enhancements to the developers efficiency when the specifications being checked gets to a certain level of complication. For specifications with only a few traces, the manual method is faster.

A scientist's aim in a discussion with his colleagues is not to persuade, but to clarify.

Leo Szilard

It is not the facts which guide the conduct of men, but their opinions about facts; which may be entirely wrong. We can only make them right by discussion.

Norman Angell

8 Discussion

The method used and tool developed in this thesis differ from most of the other consistency checking methods that exist today. The fact that the method is developed from a manual method and made computerized by letting the developed tool do the same kind of checking is somewhat unique.

The two main types of consistency checking that exist is the declarative type, which does the checking by adding rule-based definition in which the model must not violate in order to be consistent. The other main type is the transformational type which transform the model into a formal language which enforce some good properties to be held for the model to be consistent. See [30] for a survey on the different aspects of consistency in UML-based software development.

8.1 Possible approaches

The method for consistency checking used and developed in this thesis is only one of several possible approaches for consistency checking specifications. There exists different methods for comparing the behaviour found on a lifeline and the behaviour modelled in a state machine, some which are presented in this section.

8.1.1 Traces

As pointed out in section 3.2.3 on page 12 and section 3.2.4 on page 13, an interaction is an incomplete specification whose semantics are given as a pair of set of traces, while the state machine is a complete specification of either negative or positive traces.

One possible approach to finding inconsistencies within specifications could be to calculate all traces for the given lifeline and compare those to the traces of the state machine. Then all traces found within the lifeline needs to be found within the set of traces from the state machine for the specification to be consistent.

According to [36], the overall task of trace processing can be split into two phases; gathering and analyzing. To gather the traces for each specification, the routine would need to adopt some trace semantics, e.g., [35] in order to translate the specification into a set of traces, and then do some analysis of the properties to the specifications. The analysis are done in a static way and will derive properties that will hold for any execution [6].

Comparing the traces can be done in several ways. The easiest is probably to simply do textual comparison of the traces, looking at the traces for the lifeline, one at a time, trying to recognize the same trace within the set of traces for the state machine. Other methods exists for comparing traces exists, like the Iterative-Unfolding approach presented in [36].

8.1.2 Transforming lifelines to state machines

This approach differ slightly from the trace-based approach and is about transforming the lifeline into a state machine and then comparing the two state machines. In the third step in the method of specifying high-level policies with sequence diagrams in [49] they transform the sequence diagrams to state machines using operational semantics inspired by [35]. They argue that in general, a sequence diagram transformed into a state machine yields a *composite* state machine (i.e., a set of basic state machines) that contains a basic state machine for each lifeline. As we are only looking at a single lifeline at a time, we can assume that the transformation of one lifeline would result in a basic state machine.

The transformation could possibly be done by a model-to-model transformation tool, like ATL[1] or similar. As both the lifeline and state machine emerges from the same metamodel, the transformation part is quite trivial.

When the lifeline has been transfered into a state machine, we can compare the two state machines. [53] shows that comparing two state machines require special treatments due to the way that their abstract syntax are defined in the UML metamodel. Most notably is the fact that all pseudo states are of the same class and that their type is defined by setting a special PseudostateKind property. This makes comparing the differences harder as opposed to comparing two different nodes within the metamodel, e.g., comparing a state and a pseudo state. Also, there needs to be some formalization on the notion of difference calculation.

As an example, consider two state machines each having two states (see

[1] http://www.eclipse.org/m2m/atl/

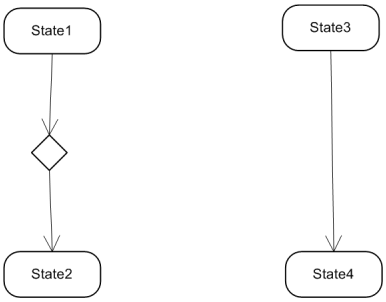

Figure 8.1: Comparing two transition paths

figure 8.1), the first one having a transition path from the one state to the other via a choice pseudo state and the second one having a transition path directly from the one state to the other. This is a pretty simple example and we argue that these two state machines corresponds as long as we are not comparing the triggers, guards or effects of the transitions, they are semantically equal but syntactically different.

Where I am not understood, it shall be concluded that something very useful and profound is couched underneath.

Jonathan Swift

9 Conclusions and future work

This section summarizes the thesis and presents the conclusions in section 9.1, achievements in section 9.2 on the next page and then finally possible future work in section 9.3 on the following page.

9.1 Conclusions

In this thesis, we investigated a method for checking consistency in UML interactions and state machines. It is a manual method which was implemented in a consistency checking tool. The interaction is the primary specification while the state machine is the concrete, runnable specification.

Firstly, we looked at a set of concepts of the consistency notion which showed that there are different types of inconsistencies to consider and also situations where inconsistencies are wanted. Then we discussed several possible approaches to how this method could be implemented, by looking at traces, model transformation and finally choosing the dynamic semantic comparison method.

We developed a routine for how the consistency checking was to be done which could be executed both manually and computerized. This routine was then implemented into a tool as a plugin for the Eclipse platform. The tool was then studied in an experiment to see how it compared to doing the consistency checking manually. Although we experienced some problems during the experiment, the tendency was that the tool was faster and more correct than the manual method, but is in need of refinement for the users to fully trust it.

9.2 Achievements

The work on consistency checking UML interactions and state machines:

- Explained the consistency notion, different kinds of inconsistencies and how it relates to UML.
- Showed how inconsistencies can be found by presenting a method which can be applied to UML interactions and state machines.
- Showed that there are certain properties, e.g., constraints, that cannot be checked at model-time, as these are runtime properties.
- Implemented the consistency checking routine as an Eclipse plugin.
- Experimented with the tool in a case study with a carefully selected group of participants.

9.3 Future work

The work with consistency in system specifications is an ongoing process within the community. During the work with this thesis, we quickly learned that there are many pitfalls one can fall into, but also many goals to reach for. Some goals have been reached, while others are still an utopia.

There are mainly two categories improvements of the work in this thesis fall into, improvements of the routine (see section 9.3.1) and improvements of the tool (see section 9.3.2 on the following page).

9.3.1 Improvement of the routine

These are possible implementable improvements to the routine:

- Handle more UML elements that can occur in both interactions and state machines, e.g.:
  - Messages with method calls.
  - Negative behaviour.
  - All combined fragment operators.
  - State machines with more than one region.
  - History states.
- Analysis of inconsistencies - "what will the consequences of fixing this this inconsistency be? how many new inconsistencies do we get?".
- Possibility of flagging certain inconsistencies as "handled", so that the routine can live with them. This might be more tool related.

- Handle more UML diagram types, or even other Domain Specific Language (DSL) models.
- Predicting what path to take within the state machine when faced with multiple possible transition paths should be improved.
- Handle more refinement concepts, like the XALT from STAIRS [26].

9.3.2 Improvement of the tool

There is a possible improvement of the alignment process. A user may not have to choose explicitly a vertex within the state machine as the tool might be able to search through the state machine, looking for the behaviour that matches the chosen event on the lifeline. This might result in several possible starting positions within the state machine, and then it would be needed to prompt the user for choosing the correct one. Whether this is a usable improvement or not is not discussed further.

The tool is in need of a better Graphical User Interface (GUI) along with better communication to the user, i.e., better formulated feedback messages. The tabs in which the user navigate in order to create an alignment, see the results etc. should be made more intuitively regarding the task at hand. The feedback messages should be spoken in a clearer voice in order to avoid misinterpreting and faulty conclusion drawn by the users.

In addition, the experiment done on the tool should be repeated with more participants in order to get higher quality of the results and to further verify the hypothesis regarding the quality of the tool.

Bibliography

[1] Egyed Alexander. Instant consistency checking for the uml. In *Proceeding of the 28th international conference on Software engineering*, Shanghai, China, 2006. ACM Press. 1134339 381-390.

[2] Egyed Alexander. Fixing inconsistencies in uml design models. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007. 1248862 292-301.

[3] Egyed Alexander. Uml/analyzer: A tool for the instant consistency checking of uml models. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007. 1248824 793-796.

[4] Zisman Andrea, Emmerich Wolfgang, and Finkelstein Anthony. Using xml to build consistency rules for distributed specifications. In *Proceedings of the 10th International Workshop on Software Specification and Design*. IEEE Computer Society, 2000. 857212 141.

[5] Nuseibeh Bashar and Russo Alessandra. On the consequences of acting in the presence of inconsistency. In *Proceedings of the 9th international workshop on Software specification and design*. IEEE Computer Society, 1998. 858308 156.

[6] Thomas Bell. The concept of dynamic analysis. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, Toulouse, France, 1999. Springer-Verlag. 318944 216-234.

[7] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of uml models. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 256–265, 2003.

[8] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 239–249, 2003.

[9] F. J. Lange Christian and R. V. Chaudron Michel. Managing model quality in uml-based software development. In *Proceedings of the 13th*

IEEE International Workshop on Software Technology and Engineering Practice. IEEE Computer Society, 2005. 1158729 7-16.

[10] Nentwich Christian, Capra Licia, Emmerich Wolfgang, and Finkelstein Anthony. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Inter. Tech.*, 2(2):151–185, 2002. 514186.

[11] L. Heitmeyer Constance, D. Jeffords Ralph, and G. Labaw Bruce. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996. 234431.

[12] Harel David. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. 34886.

[13] Harel David and Naamad Amnon. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996. 235322.

[14] Eclipse. Eclipse modeling - model development tools (mdt). http://www.eclipse.org/modeling/mdt/?project=uml2#uml2, 2007.

[15] Eclipse. Eclipse, http://www.eclipse.org, 2008.

[16] Eclipse. Eclipse model development tools (mdt) - uml2. http://www.eclipse.org/uml2/, 2008.

[17] Eclipse. Eclipse modeling framework project (emf). http://www.eclipse.org/modeling/emf/?project=emf, 2008.

[18] Eclipse. Graphical modeling framework. http://www.eclipse.org/gmf/, 2008.

[19] Eclipse. Pde introduction. http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html, 2008.

[20] M. Elaasar and L. Briand. An overview of uml consistency management - technical report sce-04-18. Technical report, 2004.

[21] et al. Erich Gamma. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[22] Holzmann Gerard. On-the-fly model checking. *ACM Comput. Surv.*, 28(4es):120, 1996. 242379.

[23] J. Holzmann Gerard. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997. 260902.

[24] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Weizmann Science Press of Israel, 2000. 903627.

[25] David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In *In Integration of Software Specification Techniques for Application in Engineering, number 3147 in Lecture Notes in Computer Science*, pages 325–354. Springer, 2004.

[26] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Stairs towards formal design with sequence diagrams. volume Software&System Modeling 00, pages 355–367, 2005.

[27] Øystein Haugen and Birger Møller-Pedersen. Javaframe: Framework for java-enabled modelling. In *ESCE 2000*, Stockholm, 2000. Ericsson NorARC 2000.

[28] Magott J. Hnatkowska B., Huzar Z. Consistency checking in uml models. In *4th International Conference on Information Systems Modelling ISM '01*, 2001.

[29] James Hobart. Principles of good gui design. http://www.classicsys.com/css06/cfm/article.cfm?articleid=20, 1995.

[30] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Louis Sourrouille. Consistency problems in uml-based software development. In *UML Modeling Languages and Applications*, pages 1–12. 2005.

[31] IBM. Plug-in development 101, part 1: The fundamentals. http://www.ibm.com/developerworks/library/os-eclipse-plugindev1/, 2008.

[32] Simmonds Jocelyn and M. Cecilia Bastarrica. A tool for automatic uml model consistency checking. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, Long Beach, CA, USA, 2005. ACM. 1101989 431-432.

[33] B. A. Kitchenham, T. Dyba, and M. Jorgensen. Evidence-based software engineering. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 273–281, 2004.

[34] Nygaard Kristen. Basic concepts in object oriented programming. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, Yorktown Heights, New York, United States, 1986. ACM. 323751 128-132.

[35] Mass Lund and Ketil Stølen. A fully general operational semantics for uml 2.0 sequence diagrams with potential and mandatory choice. In *FM 2006: Formal Methods*, pages 380–395. 2006.

[36] A. V. Miranskyy, N. H. Madhavji, M. S. Gittens, M. Davison, M. Wilding, and D. Godwin. An iterative, multi-level, and scalable approach to comparing execution traces. In *The 6th Joint Meeting on European*

software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers, Dubrovnik, Croatia, 2007. ACM. 1295035 537-540.

[37] K. Narayanaswamy and Goldman Neil. Lazy consistency: a basis for cooperative software development. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, Toronto, Ontario, Canada, 1992. ACM. 143521 257-264.

[38] OMG. Introduction to omg's unified modeling language (uml). http://www.omg.org/gettingstarted/what_is_uml.htm, 2007.

[39] OMG. Omg modeling and metadata specifications. http://www.omg.org/technology/documents/modeling_spec_catalog.htm, 2007.

[40] OMG. Uml diagram interchange (di) version 1.0. http://www.omg.org/technology/documents/formal/diagram.htm, 2007.

[41] OMG. Unified modelling language (uml) version 2. http://www.uml.org/#UML2.0, 2007.

[42] OMG. Mof specification. http://www.omg.org/spec/MOF/2.0/, 2008.

[43] OMG. Object constraint language specification v 2.0. http://www.omg.org/technology/documents/formal/ocl.htm, 2008.

[44] OMG. Omg model driven architecture (mda). http://www.omg.org/mda/, 2008.

[45] OMG. Xml metadata interchange. http://www.omg.org/technology/documents/formal/xmi.htm, 2008.

[46] Papyrus. Papyrus uml. http://www.papyrusuml.org, 2008.

[47] Dan Pilone and Neil Pitman. *UML 2.0 in a nutshell*. O'Reilly, Beijing, 2005. "A Desktop quick reference" - Omslaget.

[48] J. Robins et al. Argouml. http://argouml.tigris.org.

[49] F. Seehusen and K. Stolen. A transformational approach to facilitate monitoring of high-level policies. In *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, pages 70–73, 2008.

[50] Bernardi Simona, Donatelli Susanna, Jos, and Merseguer. From uml sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance*, Rome, Italy, 2002. ACM. 584376 35-45.

[51] Easterbrook Steve and Chechik Marsha. 2nd international workshop on living with inconsistency (iwlwi01). *SIGSOFT Softw. Eng. Notes*, 26(6):76–78, 2001. 505552.

[52] Harald Störrle. Trace semantics of interactions in uml 2.0. http://lssd.uibk.ac.at/lssd/Papers/JVLC.pdf, 2004.

[53] Kelter Udo and Schmidt Maik. Comparing state machines. In *Proceedings of the 2008 international workshop on Comparison and versioning of software models*, Leipzig, Germany, 2008. ACM. 1370154 1-6.

[54] Li Xiaoshan, Liu Zhiming, and H. Jifeng. A formal semantics of uml sequence diagram. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 168–177, 2004.

[55] Kotb Yasser and Katayama Takuya. Consistency checking of uml model diagrams using the xml semantics approach. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, Chiba, Japan, 2005. ACM. 1062829 982-983.

Appendices

A

Tool - structure

The following sections explain the implementation of the UML Consistency Checker plugin in details. Knowledge of the various Eclipse APIs[1] is needed for fully understanding

The plugin is built as a regular Eclipse plugin and can be added to the running Eclipse platform. The plugin is separated mainly into two different parts; the code specific for building the plugin/GUI and the code for the consistency check algorithm.

The different packages of the plugin can be seen in figure A.1 on the next page.

umlconsistency.checker is the main package for the consistency checking functionality.

umlconsistency.checker.commands contains the commands that is being executed when doing the consistency check on an alignment.

umlconsistency.structure is the main structure of the plugin.

umlconsistency.view contains the base classes for GUI handling.

umlconsistency.view.commands contains the commands that is being run for the different functionalities of the GUI.

umlconsistency.view.listeners contains the different listeners used in the GUI.

[1] http://www.eclipse.org

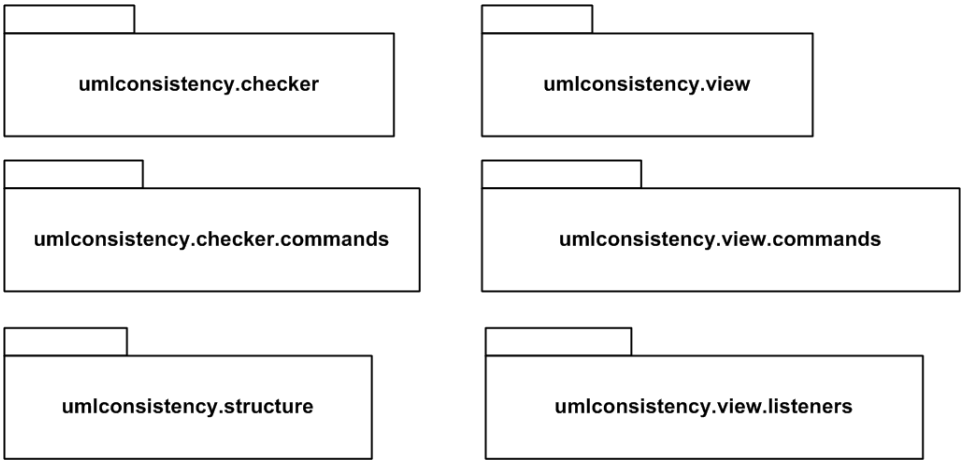

Figure A.1: The packages of the consistency check plugin

The static structure of the consistency check plugin can be seen in figure A.2 on the following page. The plugin creates a MyModel object for every model found in the domain model files in the workspace. The MyModel object holds all created alignments and its inconsistencies (if any). Both the inconsistencies and alignments has their own handler (InconsistencyHandler and AlignmentHandler) that provides common functionality to ease the work with the sets of these two important concepts.

A.1 Startup - reading domain model files

When the plugin is launched, the ModelList object initially builds a list of models found within the current workspace. It does so by looking for files with the correct file extension, given by the UML resource interface[2]. This is typically ".uml". For each domain model file found, it creates a new MyModel object (see figure A.2 on the next page) and reads the interactions and state machines of the domain model file which is then saved in the MyModel object for easy access. The domain model file is then closed and not read again until changes are made to it.

The plugin then creates the graphical user interface, as explained in detail in section A.2 on the following page.

[2] org.eclipse.uml2.uml - UMLResource

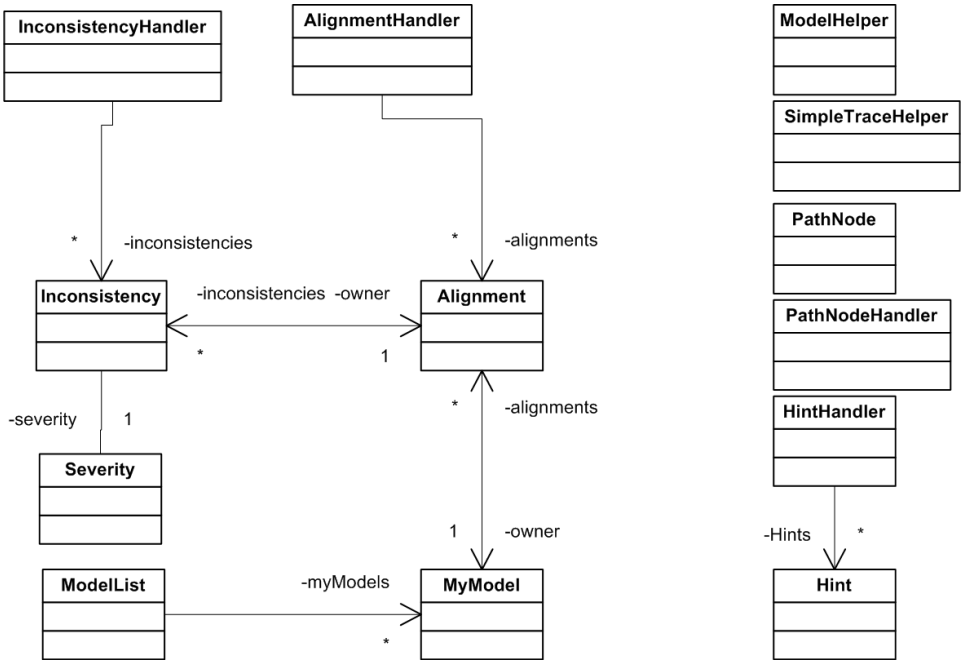

Figure A.2: The umlconsistency.structure package

A.2 Inconsistency analysis and handling - The plugin specific code

The graphical user interface (GUI) of the plugin is implemented as a view into the Eclipse workbench. It is by default located as a small window below the main editor window, although this is highly configurable and it can be moved and resized; this is the dynamics of the plugin concept of Eclipse. The graphical elements of the plugin is created using the Standard Widget Toolkit (SWT)[3] which provides graphical elements such as lists, buttons and labels.

The plugin extends the Eclipse Views library by implementing a ViewPart, adding a view that can reside in the tabbed area of the Eclipse workbench along with other view-based plugin such as the Console-, Search- and Problems view. This makes it simple for the user to interact with the plugin while she is using an editor for modelling as the views are smaller windows that are easily places on screen side by side with the editor(s). Views can be enabled by checking it in the Windows->Show View menu in Eclipse.

The view is built as a tabbed window pane. This makes it easy to split up the graphical elements into several windows and prevent too much information on each page for the user. Although the tabs do take more space on screen, the consistency check plugin does have quite an amount of data to

[3] http://www.eclipse.org/swt/

present for the user so the tabs are necessary.

The view contains five tabbed windows:

1. First tab: Inconsistencies
2. Second tab: Alignments
3. Third tab: Create alignment step 1 - model, interaction and state machine
4. Fourth tab: Create alignment step 2 - lifeline and interaction fragment on lifeline
5. Fifth tab: Create alignment step 3 - vertex

A.2.1 Presenting the inconsistencies

The tab that presents the inconsistencies is essentially a refined version of the Problems view found in the Eclipse platform. The inconsistencies tab shows inconsistencies with details and offer different actions and filtering options (see section A.2.5 on page 103). The default Eclipse Problems view has several restrictions, e.g., it only supports one single line of description of each problem and it only supports some restricted information on where to locate the problems, i.e., its scope. Due to this, there was a need for creating a Problems view-like presentation for the inconsistencies that allowed a bit more flexibility than the original one does.

Usage of the inconsistencies tab

When an inconsistency is added to the inconsistencies tab, the user has the option of highlighting these elements in the current open diagrams by toggling a check button. This is done by checking a check button on the inconsistency, on any level in the tree. The plugin will then look at each open diagram in the editor window and highlight every element found that is in the scope of the inconsistency in every diagram.

Each inconsistency has a severity value, similar to the severities found in the default markers of Eclipse shown in the Problem view. The severity values are, ranging from least severe to critical: **Info**, **Warning** and **Error**. The severity values are represented visually by intuitive graphical icons, i.e., information is shown as a blue icon, warning is shown as a yellow icon and error is shown as a red icon. When the consistency algorithm adds an inconsistency that has its severity value set to Error, it usually means that the algorithm must break because it simply cannot continue due to an incomplete specification. E.g., when a message in an interaction has no signal set, the algorithm cannot know what transition path to continue traversing within the state machine. The Warning and Info severity is used for not-so severe

inconsistencies in such manner that the algorithm actually can continue, but there is something the developer should be notified of.

The inconsistencies are ordered in a tree-wise by-model manner. Each inconsistency belongs to a model and is therefore sorted as a child to this model in the tab. Each inconsistency also belongs to an alignment and its first column contains the name of the alignment such that the user can look at the details of the alignment in the alignments tab to see the scope of the alignment. The children of each inconsistency are the model elements defining its scope.

A.2.2 Presenting the alignments

The tab displaying alignments provides functionality for saving the currently chosen alignment (which is done in the three last tabs) and removing alignments. It prevents the user from creating duplicating alignments, i.e., alignments that contains all the same chosen elements within the same model.

When a new alignment is created, an instance of the Alignment class is created (see figure A.2 on page 100) and it is added to its owning MyModel object. The alignment object then holds its scope, i.e., the chosen elements within the model and has a list of inconsistencies that is populated by the consistency check routine.

A.2.3 Create alignment step 1

The first tab when creating a new alignment has three lists. One that lists every model found in the workspace. The list of models is provided by the ModelList class, which is responsible for reading the workspace and its models and keeping internal copies. This makes the whole process of working with the model faster as opposed to working directly with the domain model file on disk as the model probably contains many more elements than the ones needed for the consistency checking.

The user must choose a model from the list before the two other lists gets populated: The second list shows all interactions found within this model. The third list shows all state machines found within this model. When the interaction and state machine are chosen, the two next tabs' lists are populated with the corresponding lifelines and vertices.

A.2.4 Create alignment step 2 and 3

When the user has chosen what model with interaction and state machine to run the consistency algorithm on the next step for the user is to align the chosen interaction and state machine. Aligning the interaction and state machine is essential as discussed in section 4.2.1 on page 22. The user must choose a lifeline beloning to the chosen interaction in addition to what vertex in the chosen state machine to start checking for inconsistencies.

The lifeline is chosen on the fourth tab and the vertex is chosen on the last tab.

When these parameters are set and the alignment is saved in the alignments tab, the plugin automatically starts the consistency check algorithm and presents the result in the inconsistencies tab.

A.2.5 Inconsistency handling - Actions and filtering options

With the possibility of having the Consistency check routine returning lots of inconsistencies, there is some need for filtering them to ease the use of the results. By right-clicking on an inconsistency, a pop-up menu appears, presenting a set of actions for the user to take.

These are the options for handling inconsistencies:

- Remove - Removes the selected inconsistencies permanently until changes are made to the model.
- Ignoration - Toggle the ignoration of inconsistencies based on their severity.
- Ignore similar - Ignore inconsistencies with the same ID (i.e., same result).
- Unignore all - Remove all ignoration, showing all inconsistencies not removed.

By removing inconsistencies, the plugin will completely remove the inconsistencies with their data structures and GUI objects. The inconsistencies will only re-appear if the model changes and the consistency check is re-run. The inconsistencies will, of course, not be shown if they do not still exists within the model.

By ignoring inconsistencies, the plugin temporarily removes the inconsistencies. Only their GUI objects are removed, making it simple to unignore the inconsistencies to show them at a later time. This is useful if the developer wants to concentrate on the most severe inconsistencies, she can then ignore the less severe ones.

These actions provide the possibility of working with the inconsistencies in a simple manner if there are many presented. E.g., the user may choose to ignore Info- and Warning-based inconsistencies and concentrate on the more severe Error-based ones.

A.2.6 Listeners

When the GUI has been built, the plugin adds several listeners to the different parts of the GUI and also a listener to the workspace itself. The former listeners are used whenever the user chooses elements within the different

lists of the tabs. The latter listener is triggered whenever there are changes, additions or removals within the current Eclipse workspace. We are mainly interested in listening for the changes that affects the model domain files.

If there are changes within the domain model files, we treat them as follows:

- Change: Update the internal representation of the domain model. Re-run the consistency check on the alignments made on this model.
- Addition: Add the domain model internally, making it available for the consistency check routine and for the user to choose when creating an alignment.
- Removal: Remove the internal representation of the domain model and remove it from the list of models in the plugin view. Also remove all alignments and the corresponding inconsistencies to the model.

A.3 Consistency checking - The consistency algorithm code

The consistency check algorithm starts when a new alignments has been created and whenever changes are made to models that contains one or more alignments.

The core of the consistency checking algorithm consists of three main parts:

1. Pre-Main loop initializing and checking of the chosen interaction and state machine.
2. Main loop that looks at the lifeline, checking interaction fragment by interaction fragment against the chosen state machine, looking for the same behaviour.
3. Post-Main loop checking.

These parts are explained in detail below.

This algorithm runs through the interaction, comparing it to the state machine. To be able to give expressive descriptions about its work, we define the following terminology:

- Current interaction fragment (int frag): The current interaction fragment on the aligned lifeline we are looking at.
- Current vertex: The current vertex in the aligned state machine we are currently looking at.

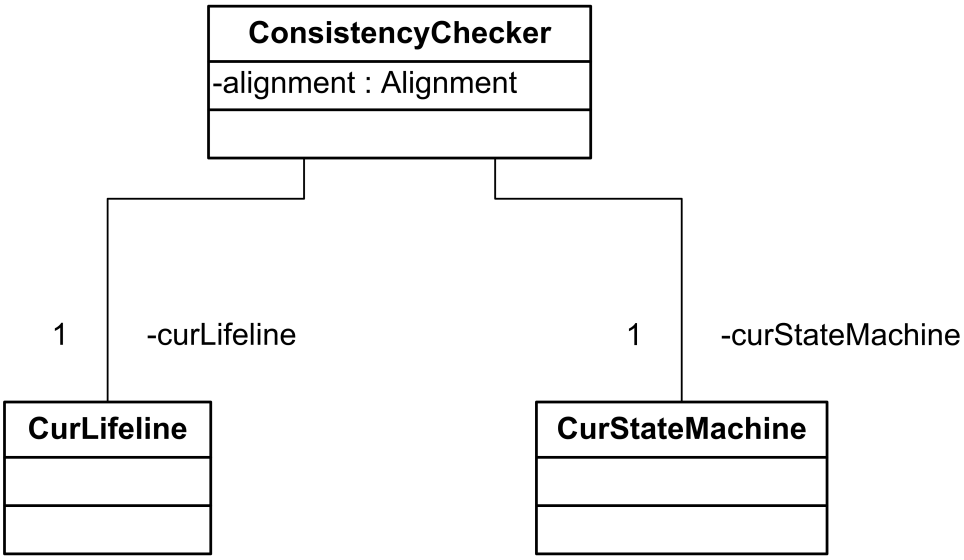

Figure A.3: The umlconsistency.checker package

- Reachable state/transition: StateB is reachable from StateA even if there one or more transition paths that possibly go via other pseudo states (junctions or choice) along the path. StateC is not reachable from StateA if the transition path goes via StateB.
- First state: The first reachable state from the initial pseudo state.

A.3.1 Pre-Main loop

When an alignment is created or modified, the plugin starts the consistency check routine on that alignment. This is done by creating an instance of the class ConsistencyChecker (see figure A.3).

Before the algorithm enters the main loop, the internal representations of the aligned lifeline (CurLifeline) and state machine (CurStateMachine) are built.

The CurLifeline class keeps a list of all interaction fragments within this lifeline and keeps track of what interaction fragment is the current one. It provides methods for, e.g., getting the first, current and next interaction fragment.

The CurStateMachine class keeps a list of all vertices within this state machine and keeps track of what vertex is the current one. It provides methods for, e.g., getting the first, current and next vertex. Other methods that are important within this class:

- findTransitionWithEffect,Trigger(Signal): Finds a reachable transition from the current vertex that has an effect or trigger containing the

given signal.

- isState,FinalState,PseudoStateReachable(): Checks if the given type of vertex is reachable from the current vertex.
- setCurVertex(Vertex): Sets the current vertex to the given vertex. If there are deferred triggers available, it will automatically forward via the transitions that gets triggered by one of these deferred triggers.

The following checkpoints are controlled pre-Main loop:

- The chosen interaction and state machine must represents behaviour of the same class. This is controlled by comparing the type of the property that the lifeline represents and the owning class of the state machine.
- If the chosen vertex is the initial pseudo state, special care must be taken when looking at the first transition:
  - There must be one (and only one) outgoing transition from an initial pseudo state. This transition cannot have triggers or guards.

Now the main loop take over the consistency checking algorithm.

A.3.2 Main loop

The main loop of the Consistency check algorithm is constructed as a loop that look at each interaction fragment on the lifeline, looking for the same behaviour in the state machine.

It looks at the event of the interaction fragment (if it is an instance of occurrence specification) and handles ReceiveSignalEvent, SendSignalEvent and DestructionEvent. If any of the former events are found and handled the main loop looks ahead and determines if the next event on the lifeline is an reception. If so, the current vertex is advanced to the next vertex. This is not done if the next event on the lifeline is the sending of a signal, because receptions corresponds to triggers within the state machine and triggers are needed to advance from one state to another.

It also handles the interaction fragment if it is an instance of an Opt or Alt combined fragment.

The functionality for handle such events are implemented as Commands (see section A.4 on page 110), they all create inconsistencies (if found) and a boolean value telling the main loop whether it can continue or must break. Often, if an inconsistency has its severity value set to Error it implicitly means that the Consistency check algorithm cannot continue and in such must return a negative value of its run, resulting in an added inconsistency.

The functionality for handling these different interaction fragments are explained in detail below in addition to challenges and other features of the main loop.

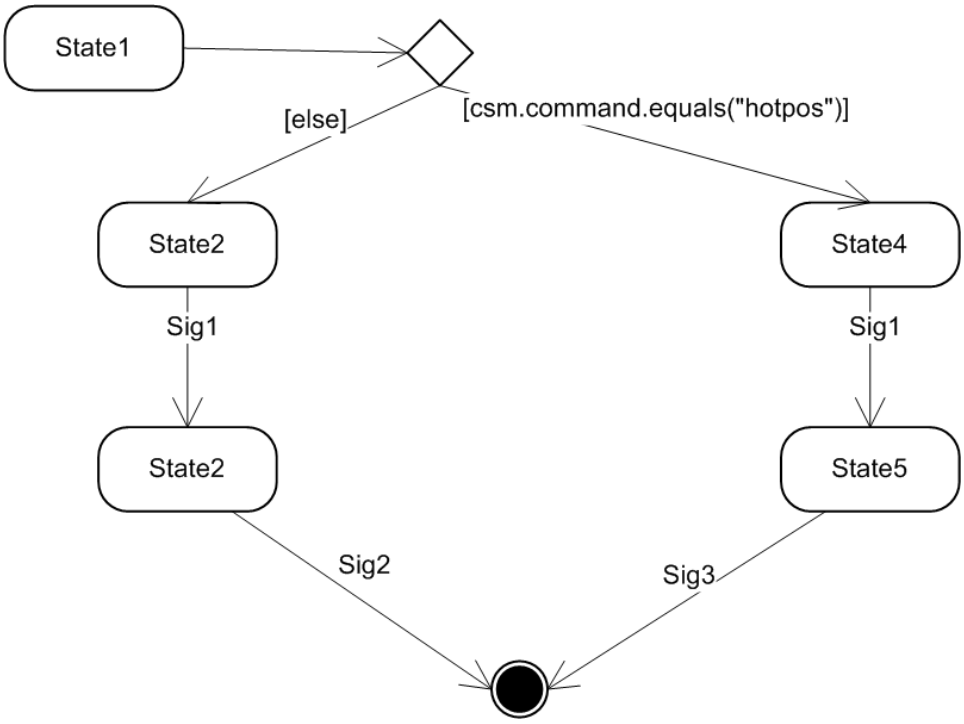

Figure A.4: Problem of choosing transition path when looking for trigger Sig1 from State1

Multiple transition paths

There exists the danger of having a non-deterministic behaviour when two possible transition paths within a state machine occurs as equally correct choices. This can be the result of a choice point that leads to at least two different states which both contains the trigger/event we are looking for (see figure A.4).

It can also occur when we have found a transition with a given trigger and this transition leads to a choice point, as we then have multiple transition paths and multiple target vertices available (see figure A.5 on the next page). As described earlier (see section 4.3.1 on page 25, guards cannot be checked at modelling time. As a result, when looking for a trigger or effect and there are more than one possible transition paths available, there is the danger of making a non-deterministic choice of what path is taken if left unhandled.

To deal with this problem, the consistency checker tool has been implemented with an AI that is designed to predict and try deciding what path to take on its own. This is done by looking further ahead on the lifeline and looking for that behaviour further ahead on the transition paths in question. If it manages to narrow it down to one single path, that one is chosen. If not, the tool question the user to make a choice on what path to choose.

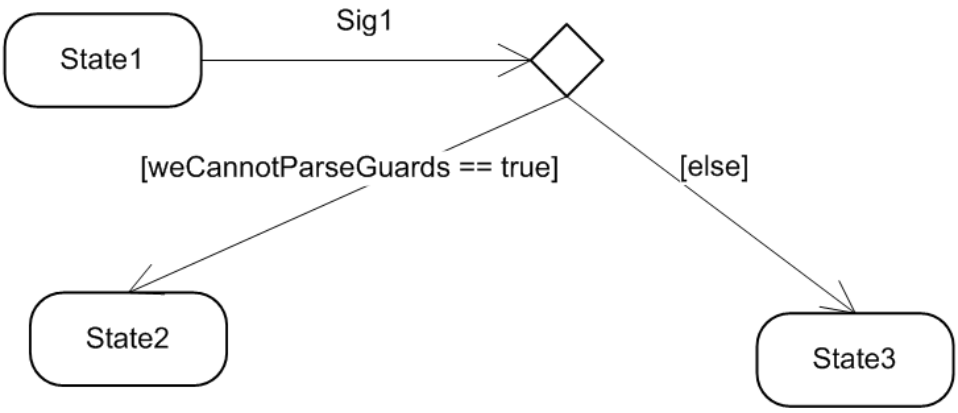

Figure A.5: Problem of choosing transition path when leaving State1 with trigger Sig1

The path chosen by the user is then stored, in order for the tool to possible use this choice later on. This functionality is implemented as a hint register which holds the previous answers given by the user when she has been questioned to make a choice between multiple transition paths. This hint register then creates a hint which either holds a chosen transition path or a chosen target vertex and it is then saved in the alignment in question. These hints are then questioned when similar situations occur.

To summarize, when the routine discovers that there are multiple transition paths available, it does the following:

1. Looks ahead on the lifeline trying to predict what transition path to chose.
2. Checks the hint for the alignment to see if it can choose a transition path based on earlier decisions made by the user.
3. If none of the above resulted in a chosen transition path, question the user to make the choice.

ReceiveSignalEvent

When the current interaction fragment is a message with the event of ReceiveSignalEvent type, the following actions are executed:

- Check that a signal is set on the message. If not, add inconsistency and break main loop.
- Check if the signal is deferred by the current state. If so, advance to next vertex and.

- Find a transition triggered by the signal received on the lifeline. If found, advance to next vertex. If not, add inconsistency and break main loop.

SendSignalEvent

When the current interaction fragment is a message with the event of - SendSignalEvent type, the following actions are executed:

- Check that a signal is set on the message. If not, add inconsistency and break main loop.
- Find a transition with an effect that sends the same signal that was sent on the lifeline. If none found, add inconsistency.

DestructionEvent

When the current interaction fragment is a message with the event of DestructionEvent type, the following actions are executed:

- Check that a vertex of type FinalState is reachable from the current vertex. If not, add inconsistency and break main loop.

CreationEvent

When the current interaction fragment is a message which is a create message, the following actions are executed:

- Check that the event is the first one on the receiving/created lifeline. If not, create inconsistency.
- Find a transition with an effect that creates the property which represents the same element as the created lifeline[4]. If not found, create an inconsistency.

Combined Fragment - Opt

When the current interaction fragment is a combined fragment of type Opt, the following actions are executed:

- Create a temporary lifeline containing the interaction fragments (from the aligned lifeline) within the operator and the ones following the combined fragment.
- Create a whole new temporary instance of the consistency checker and check the temporary lifeline and the aligned state machine.

[4] This is JavaFrame specific 5.1.5 on page 48.

- Roll back the aligned state machine to the vertex in which it was located prior to 2.

Combined Fragment - Alt

Similar as Opt, for each operator.

A.3.3 Post-Main loop

When the main loop is done, the algorithm controls whether the main loop returned a positive or negative result. If the result is negative, it means that there was an inconsistency so severe that the algorithm could not continue and the post-main loop checking immediately breaks. On the other hand, if the result is positive, it means that the main loop ran the whole lifeline from the first event and was able to finish. It might still have found inconsistencies, but those are not severe enough to break the consistency checking algorithm.

The following check points are controlled post-Main loop:

- If there are unused deferred triggers within the state machine, those are listed.

A.4 Design patterns

Design patterns can play an important role when writing efficient, reusable objectoriented programming code because they imply reusing code between projects and programmers ([21]). There are multiple design patterns available today, one of the most popular ones being the Model-View-Controller (MVC) design pattern. This is used extensively and you will find it in almost any GUI-related application. The idea is to separate objects and their functionality and controlling communications between them.

The consistency plugin has implemented the following design patterns:

- **The Command pattern**

As the user needs to choose several properties before the Consistency checking algorithm can start, the need for implementing the command pattern quickly arise when we wanted the plugin to be as self-maintained as possible and interrupt the developer as little as possible. By implementing each choice the user does within the GUI as its own command, we were able to quickly rerun all the options without having to interrupt the user. This is used, e.g., when a model is changed; instead of having the user to init the routine all over again, the plugin simply runs the command stack that contains the users' previous choices. This saves the developer time and helps to make the plugin more stand alone. We quickly discovered that by adding

this functionality to the plugin, we reduced the overhead time of the plugin a lot.

- **The Singleton pattern**

This patterns ensures that a class has only one instance by preventing others to create new instances of it. This is done by creating a global access method to the class, called getInstance(). This method creates a new instance of itself one once, and then returns the same instance each call. In addition, when writing Java it is a good practice to override the method clone() found in java.lang.Object() to prevent cloning of the singleton class, letting it simply throw an CloneNotSupportedException when called. By making a class a singleton, it can hold global data that we know will be the same when accessed from several different threads and caller classes. We then know that there is only one instance of this class, and the pointer to it is being kept by the class it self. The xxxHelper and xxxHandler classes are examples of singleton classes in the UML Consistency checker plugin. These classes operates as a common marketplace for all packages within this project, e.g., by holding references to many of the GUI objects that are being used throughout the whole project.

B

Experiment - the assignments

This section describes each assignment (10 in total) that was given to the students. Each model is based on a model[1] called "ICU", which models a cellphone-based game which allows the user to position herself. This position can be used to create an xml file with her position together with the closest positions previous stored ("hotspots"). This information can also be given per sms in return.

The following sections presents the base model and the assignments that originates from the base model.

B.1 The base model - ICU5

This is the base model for all the assignments. It contains a number of diagrams, detailing the structure and behaviour of the model. The classes of the model can be seen in figure B.1 on the following page. As the composite structure shows in figure B.2 on page 114, the ICUSystem contains three properties. Each property is modelled as a state machine:

contr:ICUController(1) is a singleton property that operate as the main controller that handles three signals:

Sms Creates a new instance of ICUProcess and forwards the signal.

PosResult and NearestHotspot Forwards the signal to the correct instance of the ICUProcess.

[1] From the course INF5150 held at the Department of Informatics, University of Oslo

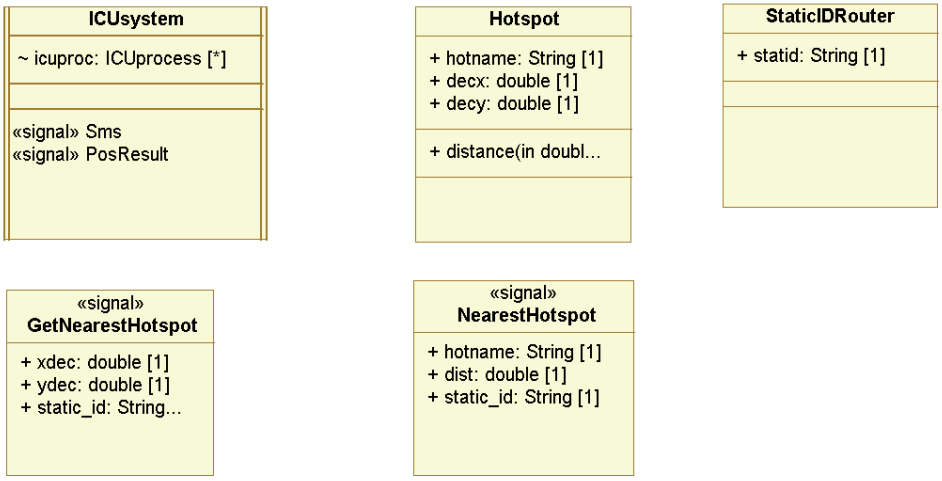

Figure B.1: The classes of the base model

icuproc:ICUProcess(*) is created for each incoming Sms to the system. It handles the Sms and engages two different actions based on the command in the Sms:

Command "hotpos" is the command that initiates the process of calculating which hotpos that is closest to the users current position. This is calculated by a sub state machine "HotPos":

1. Send a position request (signal PosRequest).
2. Receive a position result (signal PosResult) that contains the current position of the users' cell phone.
3. Calculate what hotpos is closest to the resulting position. This is done by sending a request (signal FindNearestHotpos) to the archive, waiting for the result (signal NearestHotpos).
4. Send the result to the user, or an error message if something went wrong.

Command "KML" is the command that initiates the creation of a file that contains the position of the user. The file is written in a format that is readable by the Google Earth[2] application for visualizing the position if the cell phone in question. This functionality is handled by a sub state machine called "KML".

dataproc:Archive(1) is a singleton property that holds all current hotpos positions. It also provides functionality for calculating what hotpos is closest to a given position.

[2] www.googleearth.com

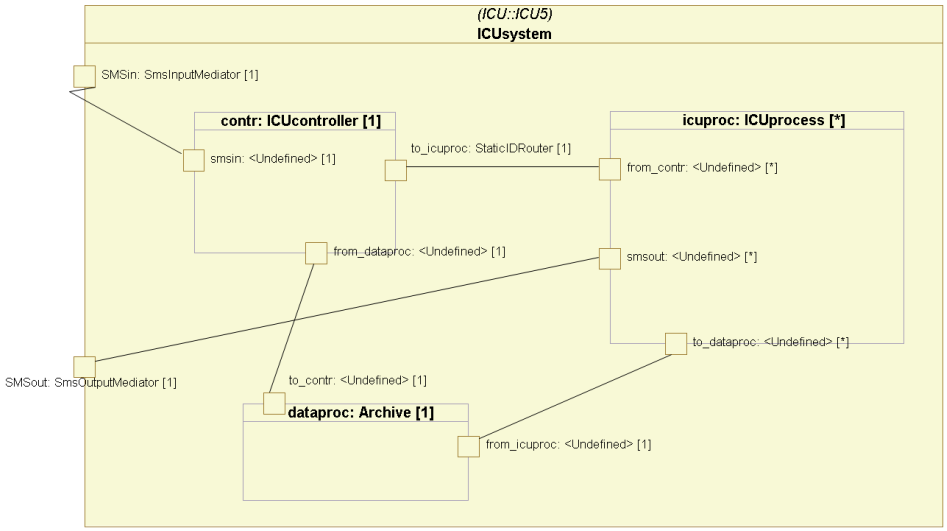

Figure B.2: The composite structure of the base model

B.1.1 State machines

This section presents the state machines of the base model.

ICUcontroller

This state machine has the responsibility of routing signals from the outside world (**Sms** and **PosResult**) and internally between properties of the composite (**NearestHotspot**). See figure B.3 on the following page. It has only one state which responds to these triggers. When an Sms signal arrives, it creates a new instance of the ICUProcess state machine and forwards the signal. When the two other signals arrive, it forwards them to the correct instance of ICUProcess.

ICUProcess

The state machine for ICUProcess contains three states, where two are submachines. See figure B.4 on the next page. When it is created, it will enter an idle-state where it awaits and handles only one trigger, namely the **Sms** signal. This signal is the parsed by the argument (command) given in the sms and the control is either passed on to the **KML** submachine or the **Hotpos** submachine if the sms had a valid command. If not, it will simply terminate.

The KML submachine handles the functionality of creating a textual xml file which contains the position of the user and its closest hotspots. See figure B.5 on page 116. It sends the **PosRequest** signal to request the

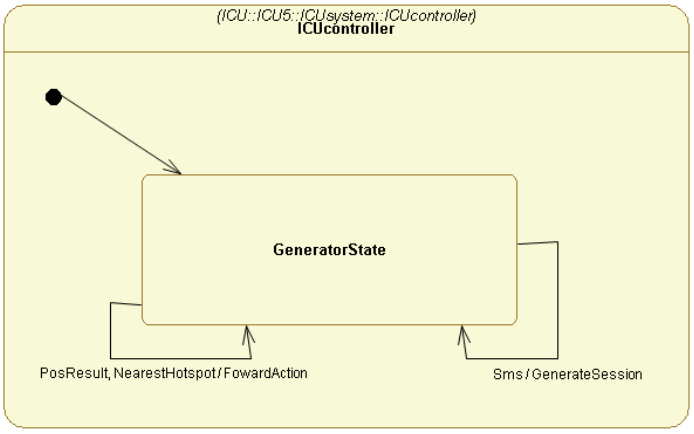

Figure B.3: State machine ICUcontroller

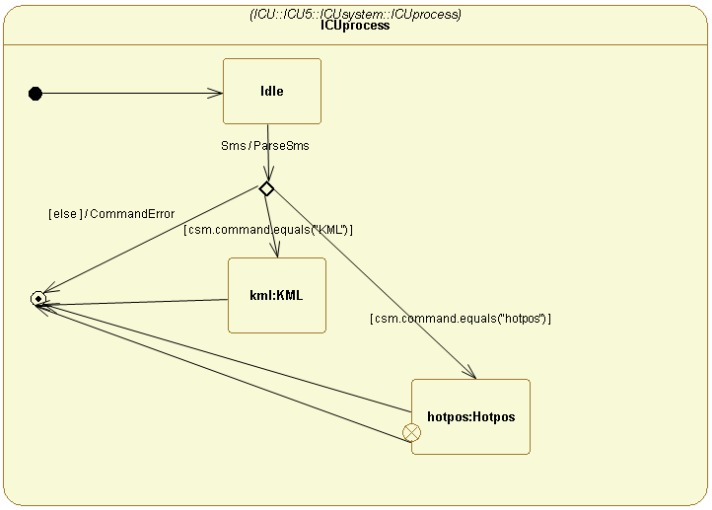

Figure B.4: State machine ICUProcess

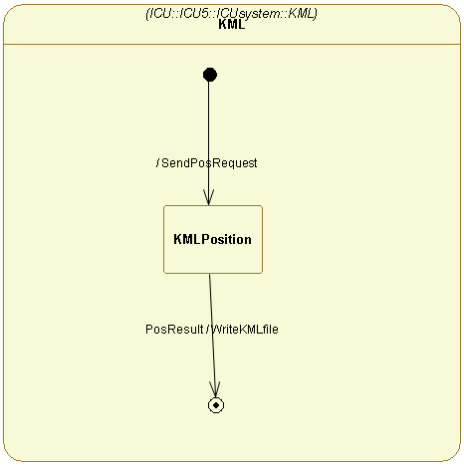

Figure B.5: State machine KML

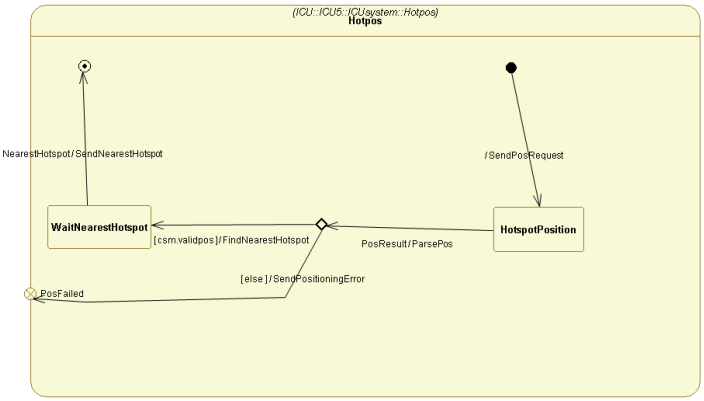

Figure B.6: State machine Hotspot

position of the user by a third-party provider. Then it awaits the result that comes with the signal **PosResult**. This result is then parsed and the xml file is created locally to the system.

The Hotspot submachine handles the functionality of sending the hotspot closest to the users current location per sms to the user. See figure B.6. It does so by sending out the **PosRequest** signal to request the users position, and then awaits the answer via the signal **PosResult**. If the resulting position validates, it asks the Archive to find the hotspot closest to this position. This is done by sending the signal **FindNearestHotspot** to *Archive* which then returns the nearest hotspot with the signal **NearestHotspot**. The Hotspot state machine then sends this information back to the user via an sms.

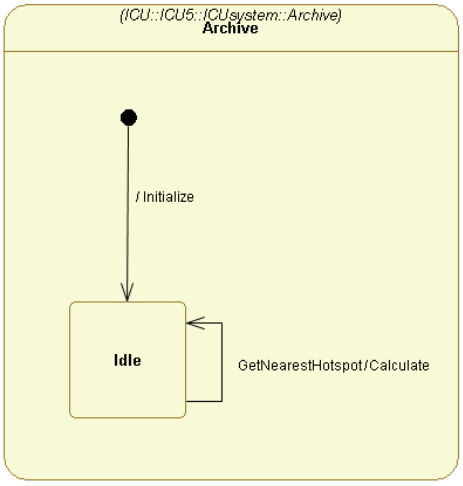

Figure B.7: State machine Archive

Archive

This state machine handles the functionality of an archive that keeps record of all hotspots registered. A hotspot is simply a place on earth given by its degrees of longitude and latitude. This is done by residing in one state only, replying to the signal **GetNearestHotspot** and return the signal **NearestHotspot** containing the nearest hotspot and its distance from the current location, which is given by the triggering signal.

B.2 Assignment 1

Interaction see figure B.8 on the following page.

Solution

- Align the lifeline contr:ICUcontroller with the state machine ICUcontroller and initial state/first state.
- The specifications are consistent.

B.2.1 Results

See figure B.9 on the next page.

B.3 Assignment 2

Interaction see figure B.10 on page 119.

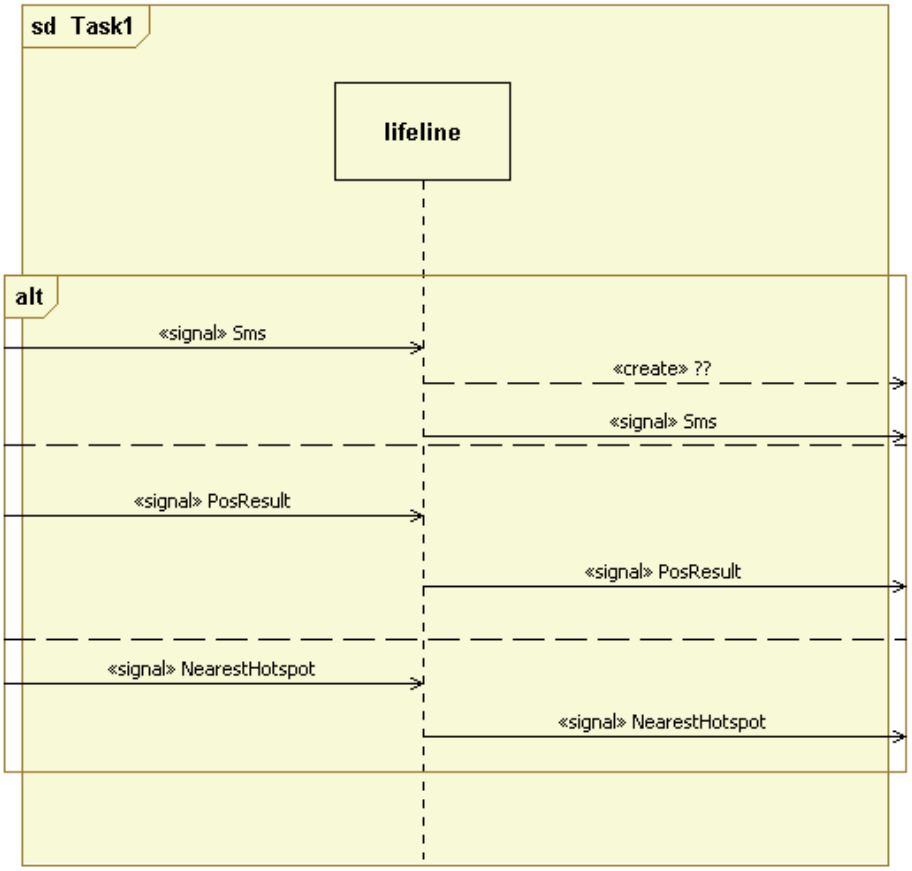

Figure B.8: Assignment 1 - Interaction

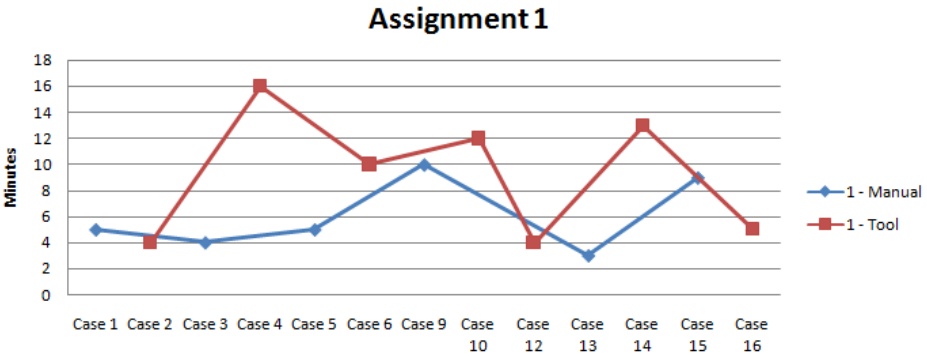

Figure B.9: Assignment 1 - Result - Time

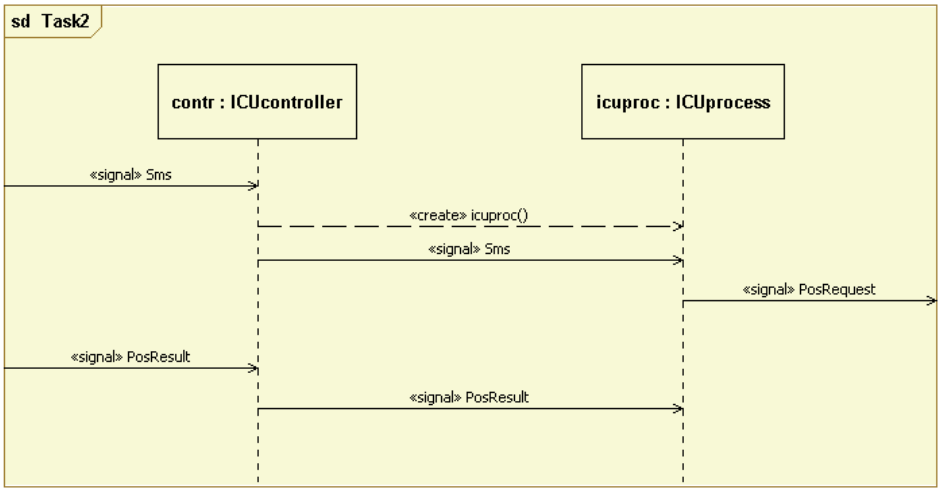

Figure B.10: Assignment 2 - Interaction

Solution

- Align the lifeline contr:ICUcontroller with the state machine ICUcontroller and initial state/first state.
- The specifications are consistent.
- Align the lifeline icuproc:ICUProcess with the state machine ICUProcess and initial state/first state.
- The consistency check tool will prompt the user to choose what path to take within the ICUProcess state machine. Both paths models the behaviour found on the lifeline so either choice will result in consistent specifications.

B.3.1 Results

See figure B.11 on the next page.

B.4 Assignment 3

Interaction see figure B.12 on the following page.

Solution

- Align the lifeline contr:ICUcontroller with the state machine ICUcontroller and initial state/first state.
- The specifications are consistent.

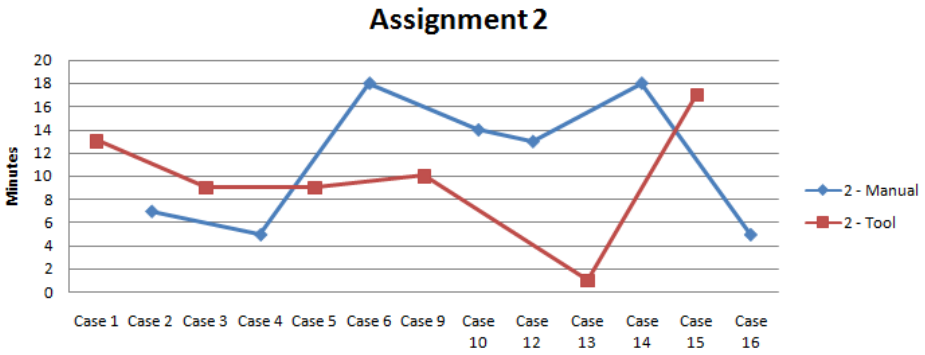

Figure B.11: Assignment 2 - Result - Time

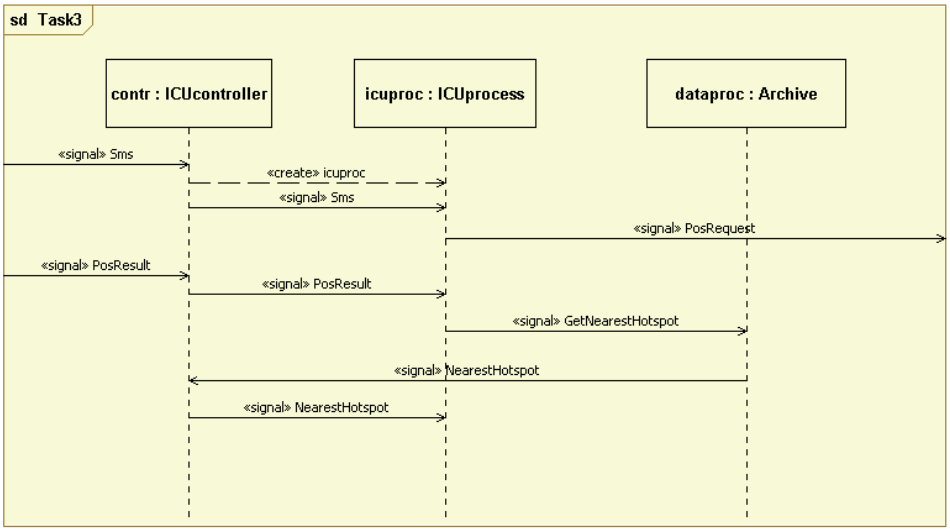

Figure B.12: Assignment 3 - Interaction

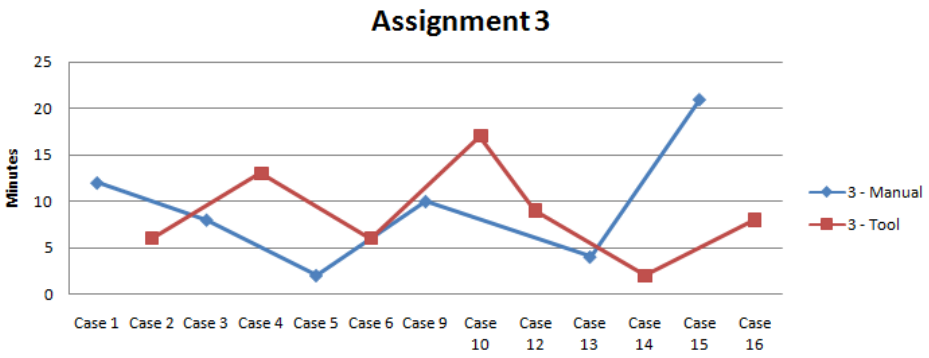

Figure B.13: Assignment 3 - Result - Time

- Align the lifeline icuproc:ICUProcess with the state machine ICUProcess and initial state/first state.
- The specifications are consistent.
- Align the lifeline dataproc:Archive with the state machine Archive and initial state.
- The specifications are consistent.

B.4.1 Results

See figure B.13.

B.5 Assignment 4

Interaction see figure B.14 on the following page.

State Machine see figure B.15 on the next page.

Solution

- Align the lifeline contr:ICUcontroller_task4 with the state machine ICUcontroller_task4 and initial state/first state.
- The specifications are inconsistent.
- There i an inconsistency based on the missing trigger on an outgoing transition from GeneratorState that triggers by **PosResult**.
- The solution is to add a trigger based on this signal to either a new transition or the one that handles **NearestHotspot**.

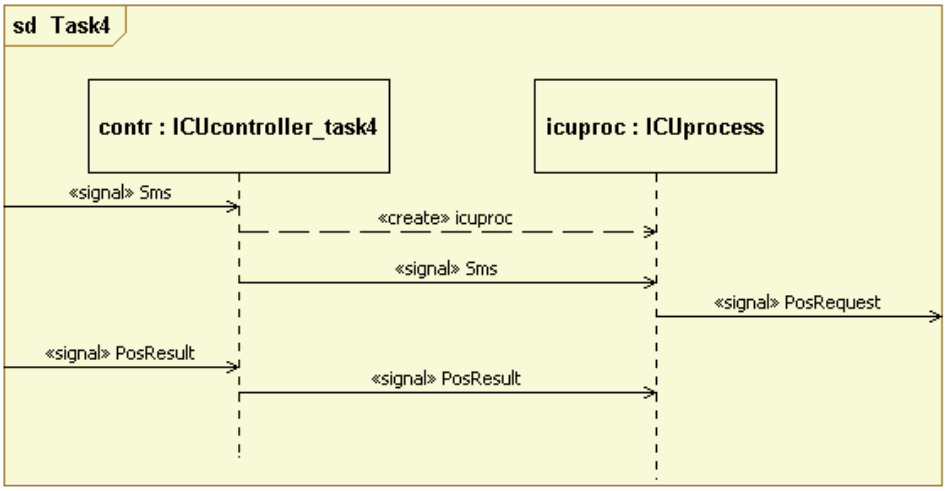

Figure B.14: Assignment 4 - Interaction

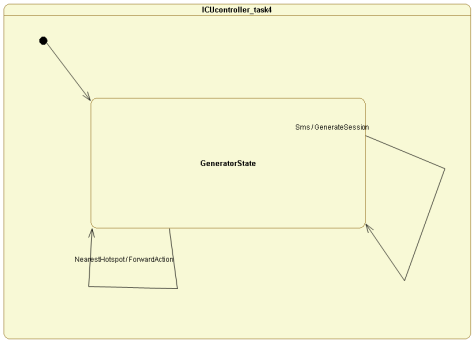

Figure B.15: Assignment 4 - State machine

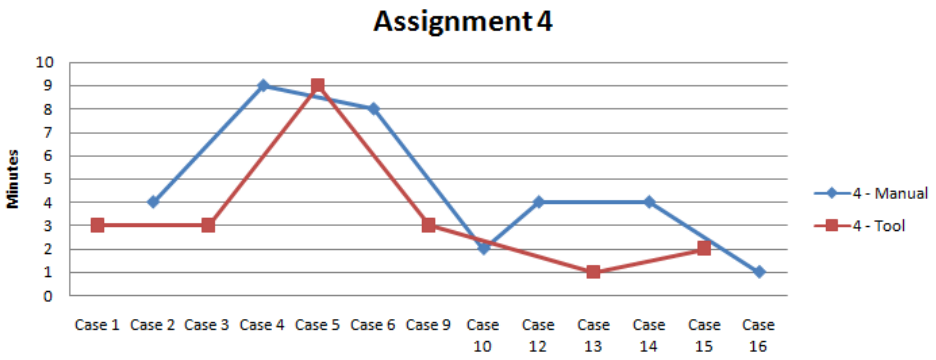

Figure B.16: Assignment 4 - Result - Time

B.5.1 Results

See figure B.16.

B.6 Assignment 5

Interaction see figure B.17 on the next page.

Solution

- Align the lifeline contr:ICUcontroller with the state machine ICUcontroller and initial state/first state.
- The specifications are inconsistent.
- There is an inconsistency based on the fact that the transition that handles the outgoing signal **Sms** does not get triggered and thus the effect will never happen.
- Align the lifeline icuproc:ICUProcess with the state machine ICUProcess and either initial state/first state, state hotpos:Hotpos or state kml:KML.
- The specifications are inconsistent.
- There is an inconsistency based on the incoming signal **Sms** that is not handled by the state machine as it is in final state.

B.6.1 Results

See figure B.18 on the following page.

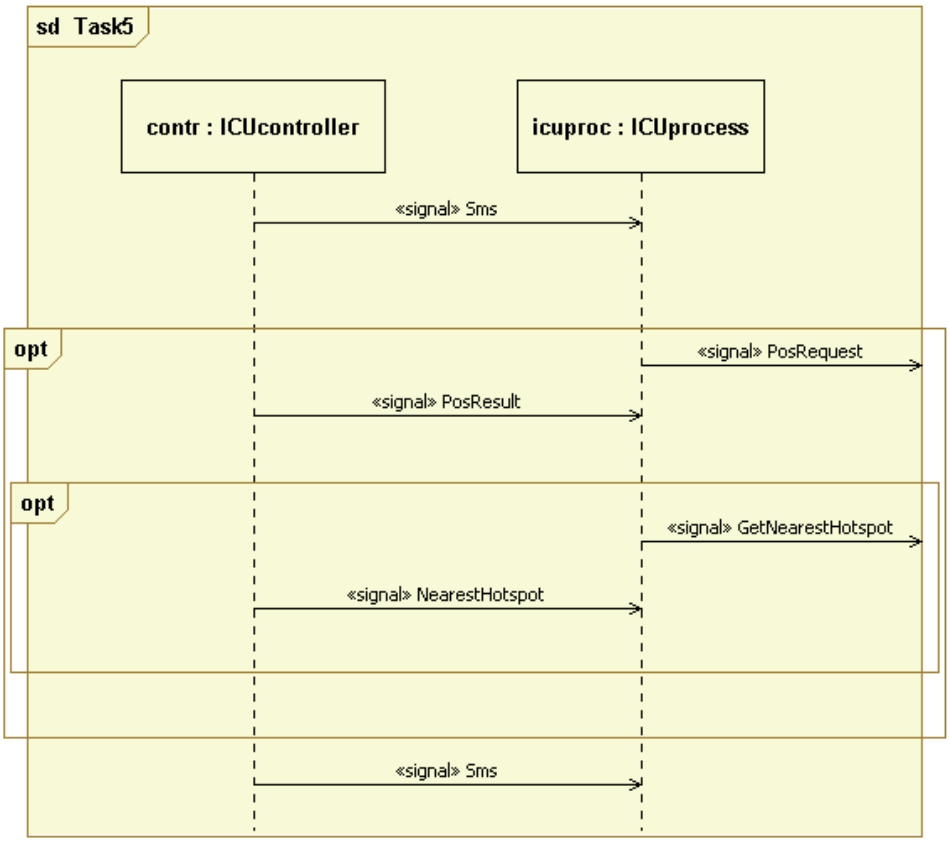

Figure B.17: Assignment 5 - Interaction

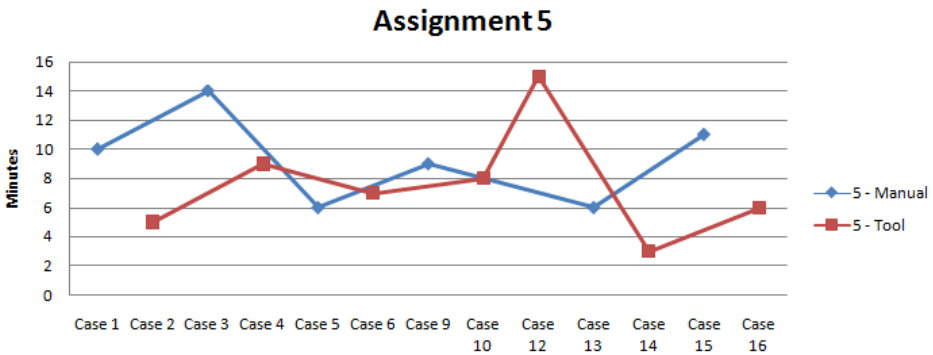

Figure B.18: Assignment 5 - Result - Time

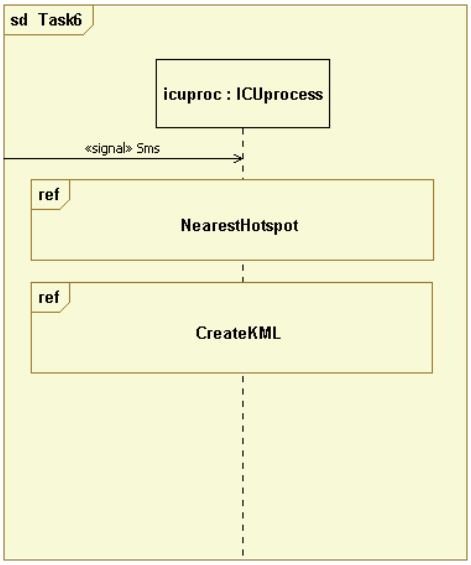

Figure B.19: Assignment 6 - Interaction

B.7 Assignment 6

Interaction see figure B.19.

Solution

- Align the lifeline icuproc:ICUProcess with the state machine ICUProcess and initial state/first state.
- The specifications are inconsistent.
- There is an inconsistency based on the second interaction use (CreateKML) on the lifeline. When the state machine has finished handling the behaviour found in the interaction use NearestHotspot, it will have reached final state and can not continue with the behaviour within CreateKML.

B.7.1 Results

See figure B.20 on the following page.

B.8 Assignment 7

Interaction see figure B.21 on the next page.

Solution

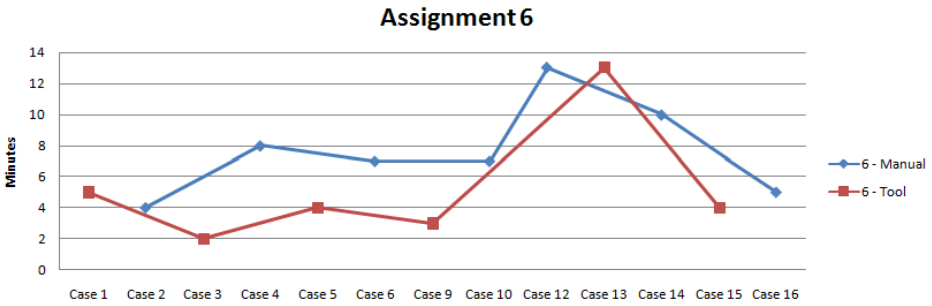

Figure B.20: Assignment 6 - Result - Time

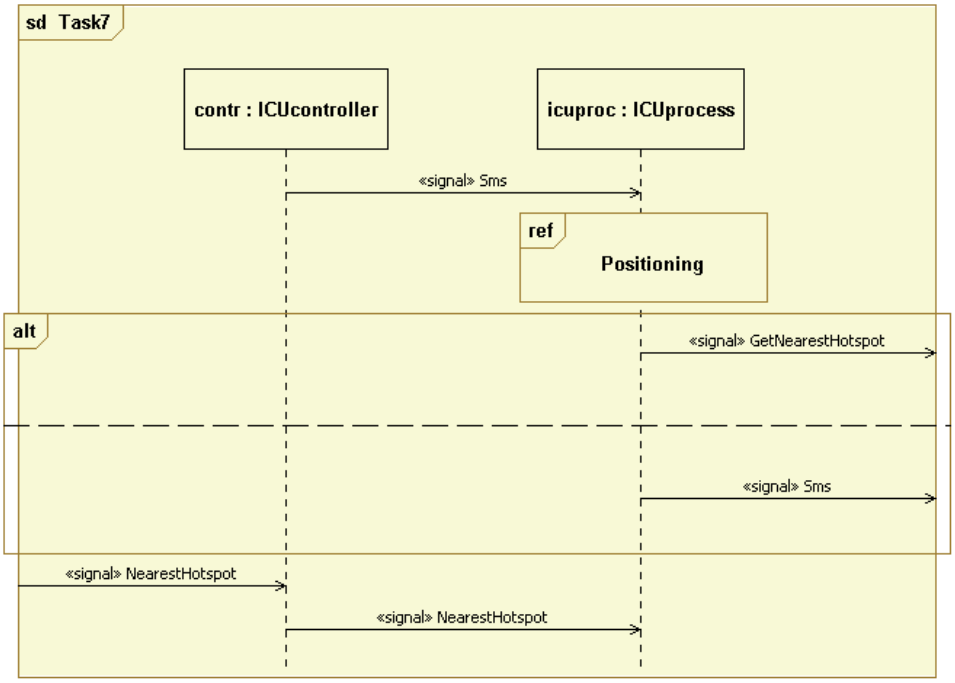

Figure B.21: Assignment 7 - Interaction

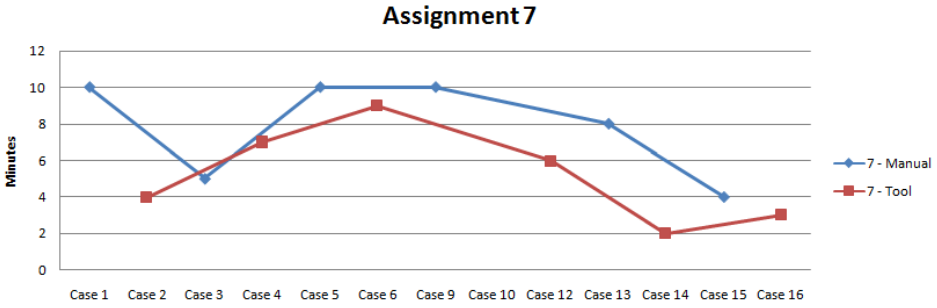

Figure B.22: Assignment 7 - Result - Time

- Align the lifeline contr:ICUcontroller with the state machine ICUcontroller and initial state/first state.
- The specifications are inconsistent.
- There is an inconsistency based on the missing trigger which should have triggered the transition on which the effect that sends the signal **Sms**.
- Align the lifeline icuproc:ICUProcess with the state machine ICUProcess and initial state/first state.
- The specifications are inconsistent.
- There is an inconsistency based on the state machine Hotpos, where it can take the path towards the exit point PosFailed when looking at the behaviour found in the second operand on the lifeline which is the sending of the signal **Sms**. It will not have a reachable transition with the effect that triggers by the signal **NearestHotspot** which is the last event on the lifeline.

B.8.1 Results

See figure B.22.

B.9 Assignment 8

Interaction see figure B.23 on the next page.

Solution

- Align the lifeline contr:ICUcontroller with the state machine ICUcontroller and initial state/first state.

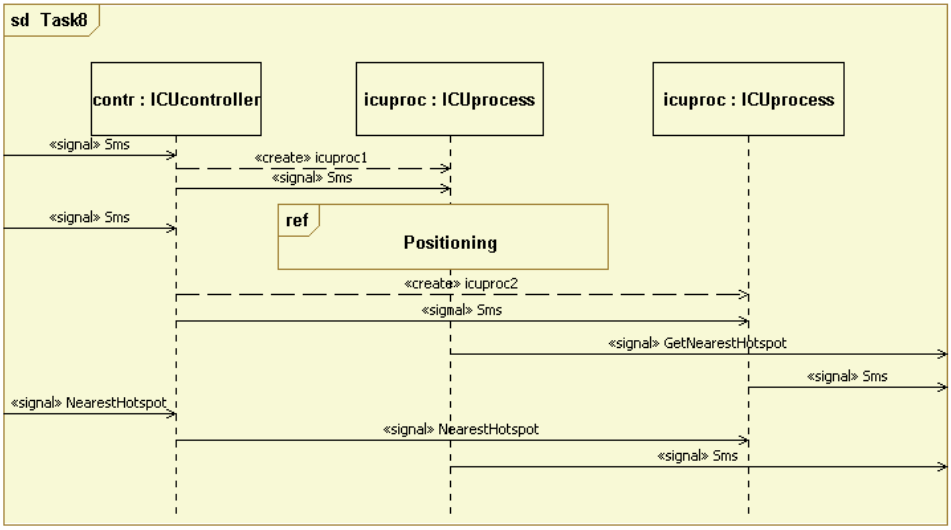

Figure B.23: Assignment 8 - Interaction

- The specifications are consistent.
- Align the lifeline icuproc1:ICUProcess with the state machine ICUProcess and initial state/first state.
- The specifications are consistent.
- Align the lifeline icuproc2:ICUProcess with the state machine ICUProcess and initial state/first state.
- The specifications are inconsistent.
- There is an inconsistency based on the fact that the state machine will enter final state when the third event on the lifeline is the sending of the signal **Sms**, which triggers the transition from the first choice point to the final state. It will not find any outgoing transitions from the final state that is triggered by the signal **NearestHotspot** which is the last event on the lifeline.

B.9.1 Results

See figure B.24 on the following page.

B.10 Assignments 9 and 10

The two last assignments are based on another model which models a simple client-server interaction.

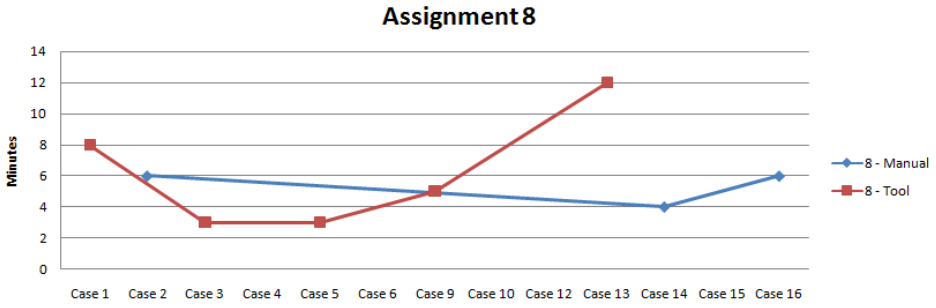

Figure B.24: Assignment 8 - Result - Time

The classes of the system is shown in figure B.25 on the next page and show that the system SystemOne consist of one server and 0..* clients and some signals that is used for interaction between the server and its clients. The composite structure shows the server and clients with their ports onto which they can send and receive the signals. The activity diagram GetConnected (figure B.27 on page 131) show the sending of the signal **Connect** to the server. The activity diagram SendAck (figure B.28 on page 131) show the sending of the signal **Accept** to the client. The activity diagram SendNAck (figure B.29 on page 131) show the sending of the signal **Reject** to the client. The activity diagram SendData (figure B.30 on page 131) show the sending of the signal **Data** to the server. The activity diagram CloseConn (figure B.31 on page 131) show the sending of the signal **Close** to the server.

The state machines used in these last two assignments model the behaviour of a typical server and client. The server (see figure B.32 on page 132) has two states, Idle and Connected, and will never terminate. It will handle a connection request by the signal **Connect** in both, but has the flaw that when rejecting a connection request it will always return to Idle. When in Connected, it can receive the **Data** and **Close** signal. When it has no more clients, it will return to Idle at the reception of the **Close** signal. The client (see figure B.33 on page 132) has only one state, WaitForAck, in which it will wait after sending the signal **Connect** a number of times (if needed) until connected with the server. When the connection request it accepted, it will send the signal **Data** 1..* times, depending on the amount of data to be sent. When there is no more data to be sent, it will send the signal **Close** to close the connection with the server and then terminate.

B.11 Assignment 9

Interaction see figure B.34 on page 133.

Solution

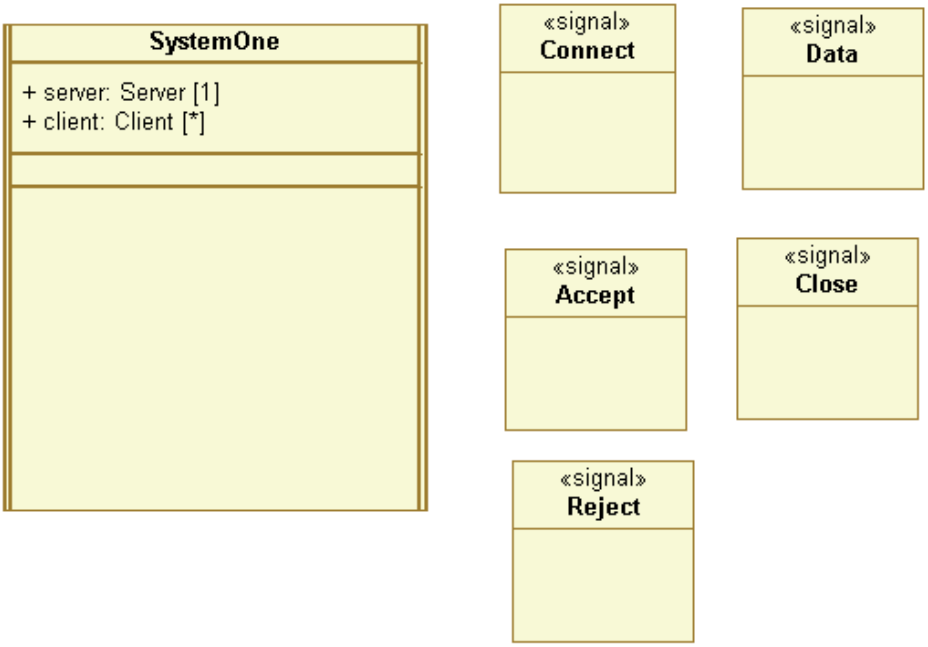

Figure B.25: Assignment 9 and 10 - Classes

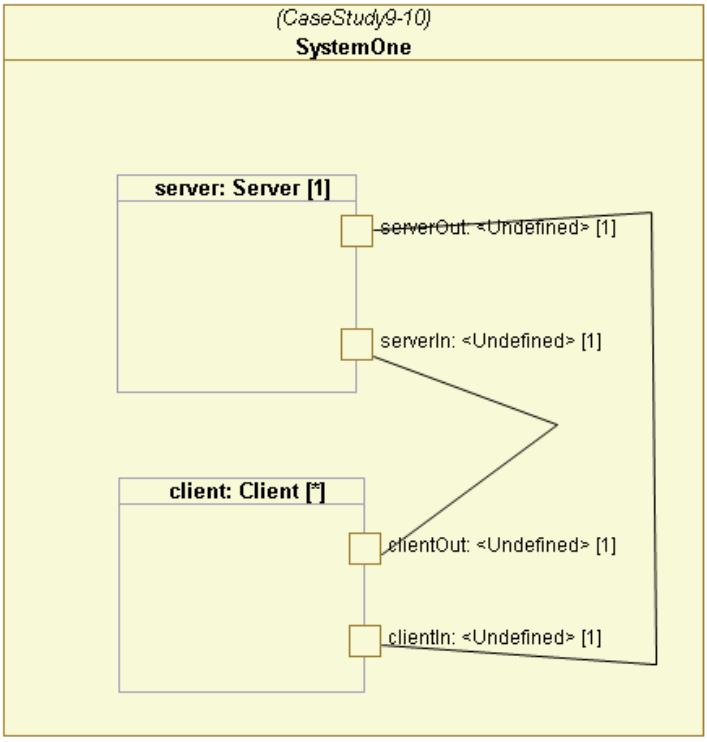

Figure B.26: Assignment 9 and 10 - Composite

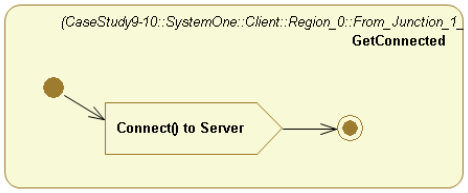

Figure B.27: Assignment 9 and 10 - Activity GetConnected

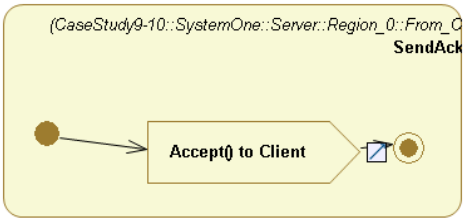

Figure B.28: Assignment 9 and 10 - Activity SendAck

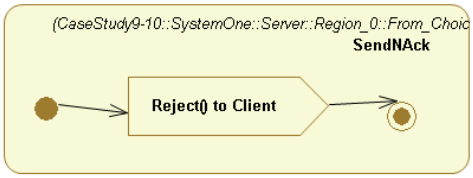

Figure B.29: Assignment 9 and 10 - Activity SendNAck

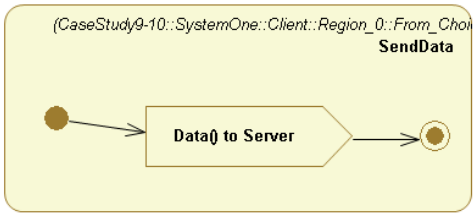

Figure B.30: Assignment 9 and 10 - Activity SendData

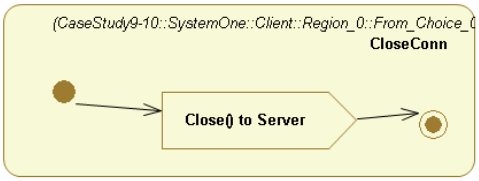

Figure B.31: Assignment 9 and 10 - Activity CloseConn

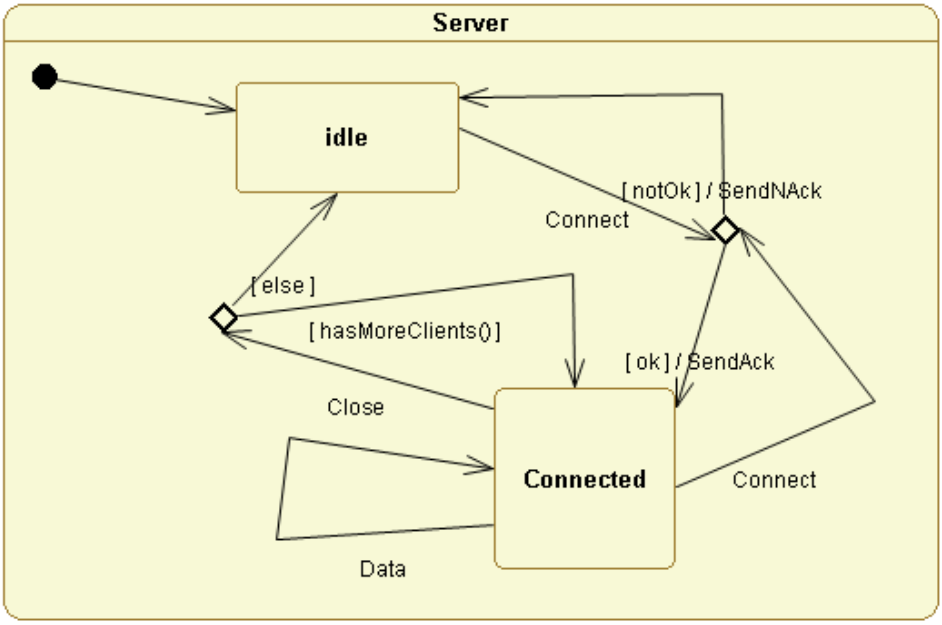

Figure B.32: Assignment 9 and 10 - State machine Server

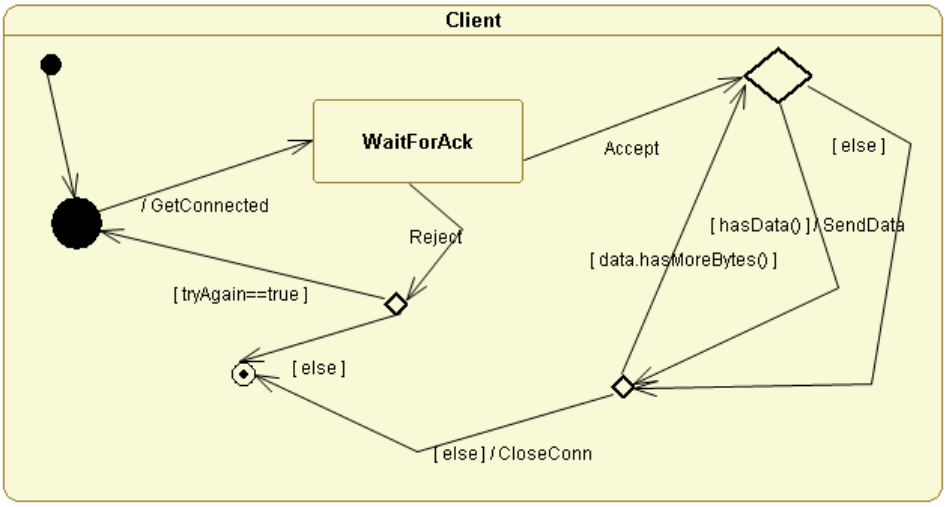

Figure B.33: Assignment 9 and 10 - State machine Client

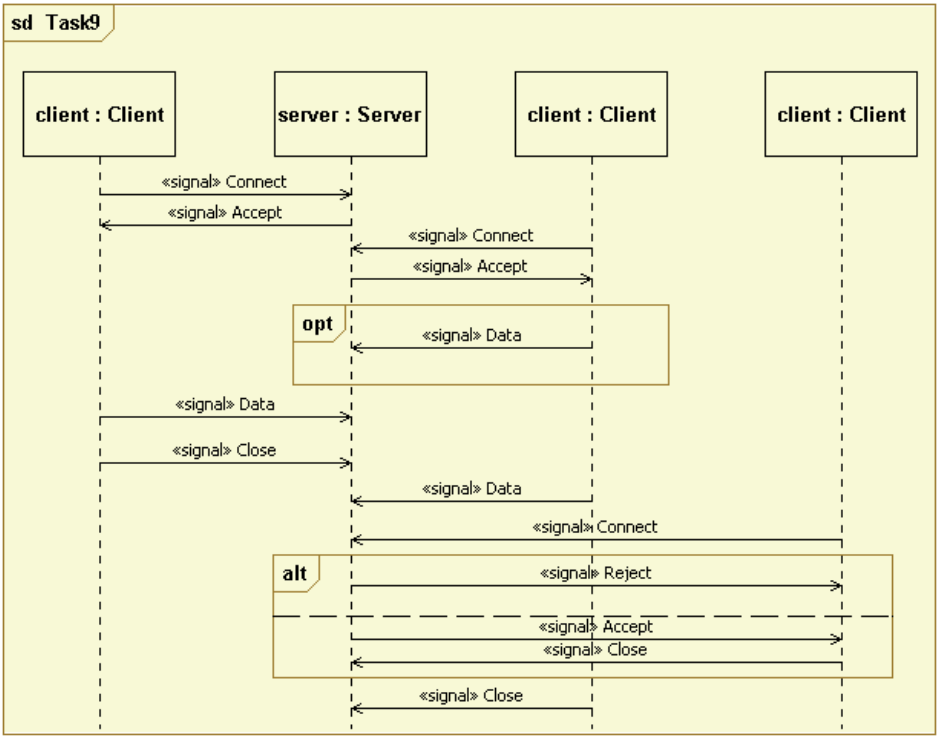

Figure B.34: Assignment 9 - Interaction

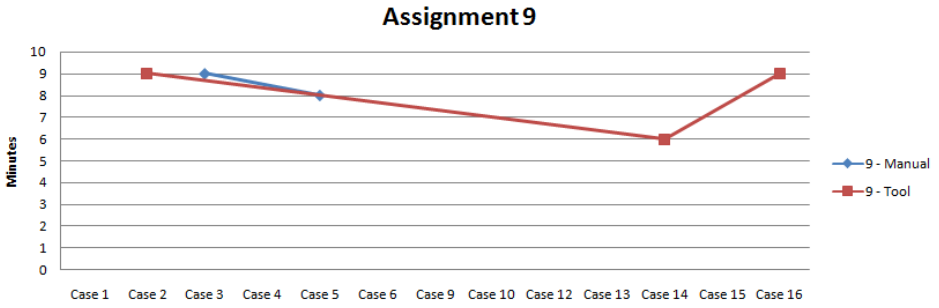

Figure B.35: Assignment 9 - Result - Time

- Align the lifeline server:Server with the state machine Server and initial state/first state.
- The specifications are inconsistent.
- There is an inconsistency based on the fact that when the server sends the signal **Reject** in the first operand of the ALT combined fragment, it will return to the Idle state and will not handle the last event on the lifeline which is the reception of the signal **Close**.
- Align the lifeline client1:Client with the state machine Client and initial state.
- The specifications are consistent.
- Align the lifeline client2:Client with the state machine Client and initial state.
- The specifications are consistent.
- Align the lifeline client3:Client with the state machine Client and initial state.
- The specifications are consistent.

B.11.1 Results

See figure B.35.

B.12 Assignment 10

Interaction see figure B.36 on the next page.

Solution

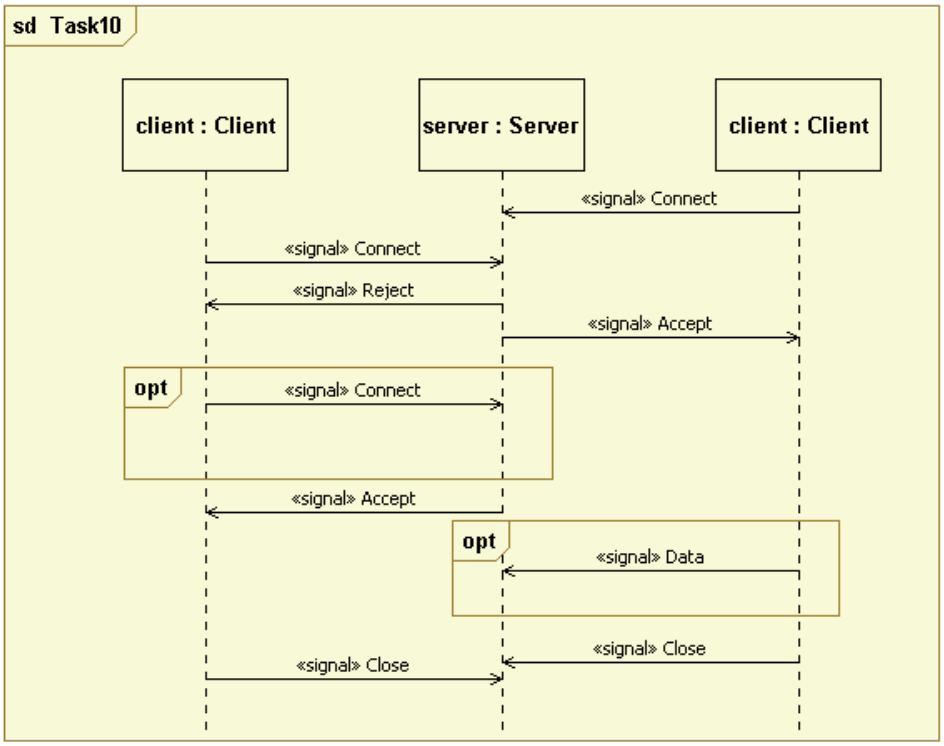

Figure B.36: Assignment 10 - Interaction

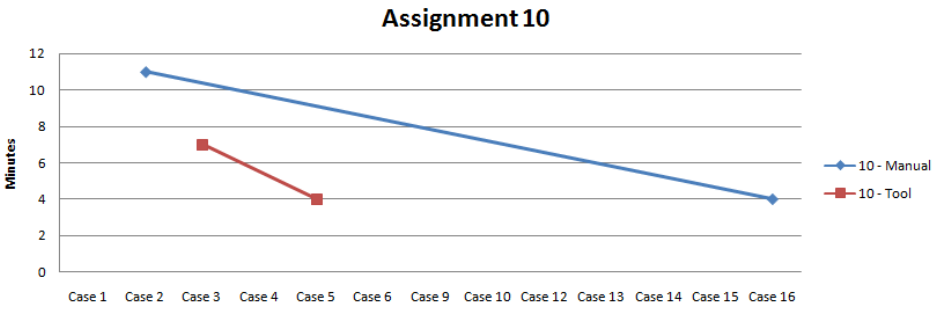

Figure B.37: Assignment 10 - Result - Time

- Align the lifeline server:Server with the state machine Server and initial state/first state.
- The specifications are inconsistent.
- Align the lifeline client1:Client with the state machine Client and initial state.
- The specifications are consistent.
- Align the lifeline client2:Client with the state machine Client and initial state.
- The specifications are consistent.

B.12.1 Results

See figure B.37.

C

Experiment - the questionnaire

This section presents the questionnaire that was given the students in the experiment when they had finished the assignments. Its purpose was to collect data regarding the consistency check tool, its usability and the users feeling of the tool. The questions were quantitatively, giving the user the possibility of answer by checking a box to rate the answers from, e.g., extremely bad to extremely good. The questionnaire consists of ten questions. The first five regards the ease of use of the tool and the last five consider the value of using the tool.

The questionnaire can be seen in table C.1 on the next page.

C.1 The overall results

See figure C.1 on page 139 for the summary table of the results.

No.	Question	Range from 1-6 where 1=very bad/no 6=very good/yes
1	How easy is it to understand the concept of consistency checking a lifeline and a state machine?	
2	How easy is it to understand the concept of the alignment of an lifeline and state machine?	
3	How easy is it to understand the graphical user interface (GUI) of the consistency checking view?	
4	How easy is it to add a new alignment in the consistency checking view?	
5	How easy is it to interpret the response/results from the plugin?	
6	Did you use the response/result from the plugin when checking for consistency?	yes/no
7	Did you trust the manual checking more than the results from the plugin?	yes/no
8	Did you *blindly accept* the results from the plugin or did you *double check* the answers manually?	blindly accept/ double checked
9	Do you think you could make use of the tool in the course	yes/no
10	Overall value of the consistency check tool	
	Comments:	

Table C.1: The questionnaire

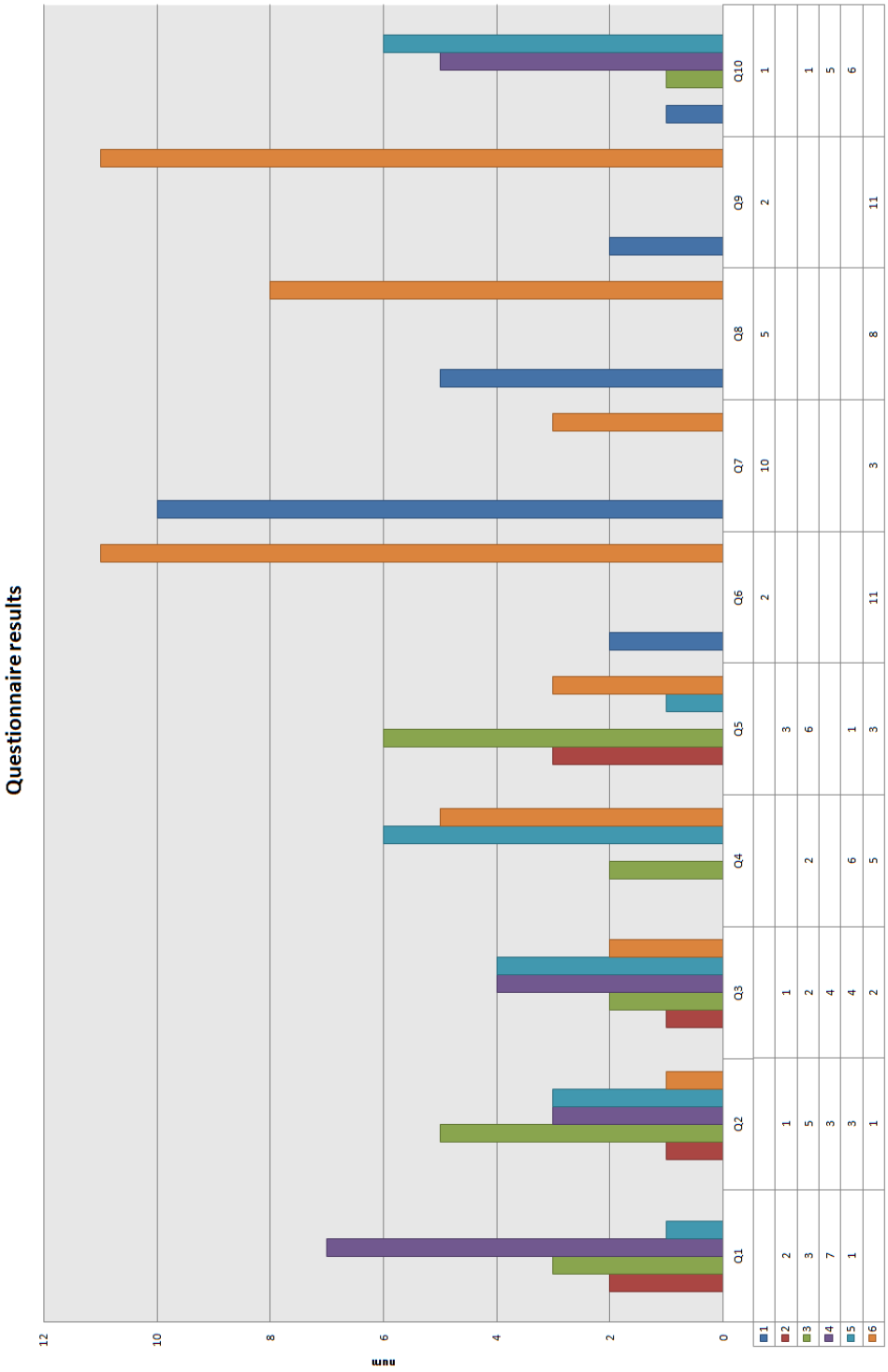

Figure C.1: Questionnaire - Result table